VU VRIJE UNIVERSITEIT AMSTERDAM

VU Research Portal

Building a Reliable Storage Stack

van Moolenbroek, D.C.

2016

document version
Publisher's PDF, also known as Version of record

Link to publication in VU Research Portal

citation for published version (APA)
van Moolenbroek, D. C. (2016). *Building a Reliable Storage Stack*. [PhD-Thesis - Research and graduation internal, Vrije Universiteit Amsterdam].

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 05. May. 2023

Building a Reliable Storage Stack

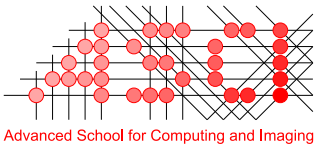
Ph.D. Thesis

David Cornelis van Moolenbroek

Vrije Universiteit Amsterdam, 2016

vrije Universiteit amsterdam

This work was supported by the European Research Council Advanced Grant 227874.



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 355.

Copyright © 2016 by David Cornelis van Moolenbroek.

ISBN 978-94-028-0240-5

Cover design by Eva Dienske.
Printed by Ipskamp Printing.

VRIJE UNIVERSITEIT

Building a Reliable Storage Stack

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op maandag 12 september 2016 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

David Cornelis van Moolenbroek

geboren te Amsterdam

promotor: prof.dr. A.S. Tanenbaum

To my parents

Acknowledgments

This book marks the end of both a professional and a personal journey–one that has been long but rewarding. There are several people whom I would like to thank for accompanying and helping me along the way.

First and foremost, I would like to thank my promotor, Andy Tanenbaum. While I was finishing up my master project under his supervision, he casually asked me "Would you like to be my Ph.D student?" during one of our meetings. I did not have to think long about the answer. Right from the start, he warned me that I would now have to conduct original research myself; only much later did I understand the full weight of this statement. Throughout the years, Andy's supervision has been fairly hands-off, but he has also been truly supportive especially at key moments. As a result, I learned a lot, but I was also able to see it through to the end. I am incredibly proud to have worked as a Ph.D student under Andy's supervision, and incredibly grateful that he guided me through the process.

Among Andy's contributions, perhaps the most important one was to suggest that I collaborate with Raja Appuswamy. As a result, I had the opportunity to work with Raja on his great new idea at the time, which turned into Loris and ended up being the ground work for this thesis as well. During our collaboration, I learned so many things from Raja–ranging from the more practical side of doing good research to various aspects of Indian culture. Working with him has always been pleasant and easy, and outside work I have been lucky enough to share several memorable adventures with him. Now that it is my turn to face the guillotine, I could not wish for a better paranymph!

I would also like to thank Dirk Vogt, whom I am honored to have as my other paranymph. With his inspiring enthusiasm, broad interests, challenging ideas, and a great gift for generating entropy, he has been a key figure in the later years of my Ph.D and is a major part of the reason that I miss going to the VU on a regular basis.

During my years as a Ph.D student, I have had the fortune to share office space and time with all of the researchers in Andy's MINIX 3 group: Jorrit Herder, Mischa Geldermans, Raja Appuswamy, Cristiano Giuffrida, Tomáš Hrubý, Lorenzo Cavallaro, Dirk Vogt, and Erik van der Kouwe. From detailed technical discussions to pizza-and-movie nights, it has always been fun. The programmers deserve mention in the same breath: Ben Gras, Philip Homburg, Kees Bot, Arun Thomas, Thomas Veerman, Gianluca Guida, Kees Jongenburger, and Lionel Sambuc. They tolerated that I regularly put on a programmers' hat as well, making my time as Ph.D student much more enjoyable even if longer!

I would like to thank several others. The various students that contributed to MINIX 3, and the Loris project in particular, including Arne Welzel, Richard van Heuven van Staereling, and Sharan Santhanam.

The *Dutch lesson group*: Ana-Maria Oprescu, Corina Stratan, Albana Gaba, Philip Homburg, Dirk Vogt, Daniela Remenska, and later various others. I will always cherish the amazing, cultural, crazy evenings and nights I shared with these fantastic people.

The larger VU computer systems group for many fun days and evenings in and around the coffee room; Kaveh Razavi and Caroline Waij in particular.

Arjen van Deutekom, for always being a good listener and a welcome source of interesting discussions, good advice, and comic relief.

The *moo team*, for being somewhat of a home to me, even if virtual. Especially Oliver *[Redacted]* and Loren Segal, for enduring all my stories about work frustration, helping me out with a few important issues, and providing many distractions.

My talented sister, Eva Dienske, who always pushes me to get the most out of myself, and whose work I am proud to have on the cover of this thesis. My parents, Jaap van Moolenbroek and Joke Putters, for their undying support, encouragement, understanding, and love.

Finally, I would like to thank the members of my reading committee: Herbert Bos, Cristiano Giuffrida, Jorrit Herder, Spyros Voulgaris, and Carsten Weinhold, who have taken the time to read my thesis and provide valuable feedback.

David van Moolenbroek
Hilversum, The Netherlands, June 2016

Contents

1 General Introduction

In an ideal world, computer systems would always operate correctly. Practice is different however: it is well known that both the hardware and software components of computer systems are subject to several forms of failure. There are many causes for failures: a poor hardware manufacturing process, physical wear, loss of power, cosmic radiation, programmers introducing software bugs, and so on. The resulting failures may subvert the operation of a computer system, thereby not only making its functionality unavailable temporarily, but possibly also causing *data loss*, which often results in more lasting damage. A plethora of real-world cases have shown that these dangers are real and can lead to significant monetary losses–or worse [68, 79, 81, 123].

As a result, there is an active field of research on computer systems dependability, and specifically **reliability**, aiming to ensure that computer systems continue to operate correctly in the presence of *faults* (causes of failures) [14]. Since faults can often not be prevented, a common goal is *fault tolerance*: a system's ability to avoid that faults turn into full-scale system failures. Fault tolerance consists of *detection* of manifested faults (*errors*) and subsequent *recovery* of the system. Overall, these reliability improvements provide the benefit of avoiding costly disasters in the infrequent but important cases that errors occur.

In all but the most trivial cases, fault tolerance comes at a cost as well. For error detection, this cost is paid in the form of more expensive verification of correct operation; for recovery, it is typically paid in some form of redundancy. In concrete terms, the cost can take the form of more hardware, lower system performance, higher storage and memory usage, increased software complexity, and so on. The cost has to be paid even when the extra reliability goes unused. In general, stronger reliability guarantees tend to come at a yet higher cost, thus yielding a whole spectrum of possible reliability improvements.

In this thesis, we aim to advance the state of the art of computer systems fault tolerance by making reliability improvements in one particular area that we believe is in particular need of such improvements: the operating system storage stack. We will now explain how we arrived at this area.

First of all, we must consider the cost aspect of our improvements. After all, even though the benefits of improved reliability are relevant to the owners of all computer systems, low or modest cost is essential for the possible real-world adoption of any solution. As a result, we only consider the **software** side of computer systems. While improvements in hardware are often able to provide good guarantees, changes on the hardware side also tend to be expensive. In addition, hardware improvements typically do not become available for computer systems that are already in operation, thus eliminating a large potential target group of systems that would benefit from improved reliability. In contrast, software-based improvements may not always be able to provide the same guarantees of hardware solutions, but they also tend to be less expensive, and can be applied on existing computer systems as well.

Within the software stack, we assert that the primary focus of reliability research should be the **operating system** layer, for two reasons. First, the operating system is the main *single point of failure* in the software stack. Logically positioned between the hardware and the running applications, the operating system controls the hardware resources, and exposes these as high-level abstractions to the applications. The typical operating system creates a separate *failure domain* for each running application, allowing multiple applications to run in isolation from each other. Thus, a reliability problem in a particular application affects only that application. However, a reliability problem in the operating system may end up affecting all the running applications as well.

Second, only the operating system can ensure that reliability problems, both in the underlying hardware and within itself, are not fatal to the operation of the computer system, and to running applications in particular. Operating systems have fulfilled this role for decades–for example, reportedly, about half of the code of the Multics operating system was related to error recovery [155]. However, over time, our understanding of the ways in which computer systems can fail has improved, and this in turn has revealed that today's operating systems do not cope well with several real-world reliability problems. The result is a "gap" between the level of reliability offered by contemporary operating systems, and the reliability threats that these operating systems are actually up against.

Within the operating system, we argue that the component most in need of closing this gap is the **storage stack**. The storage stack consists of the software layers collectively responsible for all aspects of data storage in the system, ranging from managing low-level storage hardware to exposing high-level storage abstractions such as files and directories to applications. The storage stack stands to be affected by several different external reliability problems: storage device failures, whole-system failures, and memory corruption. As we will argue, these problems are currently not handled satisfactorily. In addition, modern storage stacks often consist of

hundreds of thousands of lines of code optimized primarily for high performance, which makes it likely that the storage stack contains many software bugs, each of which has the potential to bring down the entire system or cause data loss. At the same time, from an application perspective, the storage stack is generally expected to perform operations without reporting exceptional failures–in contrast to the network stack, for example. Thus, if the storage stack itself does not adequately shield applications from reliability problems, there is little hope that applications will be able to recover from these problems themselves.

Therefore, in this dissertation, we aim to improve the reliability of the operating system storage stack. Our goal is to provide better protection from reliability threats for both applications and data, while also keeping in mind the cost aspect; therefore, we specifically explore software solutions that have low or modest overhead. In the remainder of this chapter, we first elaborate on the exact problems we aim to address (Sec. 1.1), then describe our research approach and the research questions we aim to answer (Sec. 1.2), give a broad overview of our research (Sec. 1.3), and finally give an outline of the rest of this thesis, including per-chapter contributions (Sec. 1.4).

1.1 Problems

We now describe the four storage stack reliability problems that we consider in this dissertation. The first two problems involve particular failures that by themselves are not fatal to the long-term operation of the computer system, but may result in subsequent faults being introduced into the system. The last two problems concern faults themselves. In all cases, if not taken care of, these faults may end up causing more serious failures down the line, including corrupted data being passed to applications and permanent data loss.

The first problem is **storage device failures**: failures that occur within the storage device hardware–typically hard disks. Traditionally, the failure mode of storage devices has been modeled as *fail-stop*: either the device functions correctly, or it reports failure to the operating system. However, research has shown that hard disks can fail in various other, *partial* ways [16], resulting in *silent data corruption*: corruption of data without any indication that corruption took place. Specifically, malfunctioning disk hardware or firmware may cause lost, misdirected, and torn writes, where write operations are (silently) not performed, performed at the wrong device location, or completed partially, respectively. In addition, storage devices may be subject to *bit rot*, where the contents of a data block become corrupted due to decay of the storage medium. As a result, the storage device ends up containing faulty data. Analysis of disk errors in large installations shows that such failures indeed occur in practice [16, 80]. Furthermore, the use of a software implementation of RAID (Redundant Array of Inexpensive Disks [113]) within the traditional storage stack may exacerbate this problem: for legacy reasons, the RAID layer operates transparently below the file system layer, and this arrangement creates an *information gap*

between the two layers [33]. This information gap may lead to undetected propagation of corruption, subverting the guarantees supposedly offered by the RAID layer and resulting in possible destruction of data [88]. Thus, it is key that silent storage device failures be detected before the resulting faults can propagate and cause more damage. Once data corruption is detected, recovery requires a redundant, good copy of the data. However, ideally a storage system continues the best it can even in the light of partial, permanent data loss.

The next problem is **whole-system failures**, or *system crashes*: unexpected shutdowns of the entire computer system. One cause of such an event is a power failure, although it may also be caused by a hardware failure or a failure in a critical part of the operating system. Whole-system failures can often not be prevented, but are also transient. Thus, from a reliability perspective, the focus should be on the consequences. In particular, a system crash may lead to the situation that some but not all scheduled data changes have yet been written to stable storage at the time of the crash, which may result in an inconsistent state on the storage devices. Thus, whole-system failures may lead to corruptions in stored data, and these inconsistency faults may lead to a variety of failures upon a subsequent system reboot, including the worst case: loss of data, including data that was *not* being updated at the time of the original failure. Therefore, the recovery procedure must eliminate the possibility that any such faults remain. The problem of whole-system failures is well known, and several different solutions have been proposed and implemented in file systems in order to allow recovery to a consistent state, including journaling [53], logging [112], copy-on-write [66], and soft updates [43]. However, as we will show, a proper solution for storage device failures in fact creates a new challenge for whole-system failure recovery.

Another problem is **memory corruption**; in particular, transient bit changes (*soft errors*) in the computer system's main dynamic random access memory (DRAM or RAM), as a result of external factors such as cosmic rays. Previous research has yielded varying estimations for the frequency of soft error occurrences in DRAM [95, 124, 145]. While error-correcting code (ECC) RAM is able to recover from the majority of such cases, many computer systems today are not equipped with ECC RAM because of the added cost. Memory corruption is relevant for the storage stack in particular: on typical modern operating systems, the largest fraction of RAM not used by applications is used by the operating system for the purpose of caching storage device contents. Thus, a random bit change in memory is relatively likely to affect the storage stack's cache. In addition, storage cache memory may end up being read and modified by any application in the system, so memory corruption in the storage cache may arbitrarily affect the entire computer system. Only the storage stack is in the the position to mitigate this problem, by detecting corruption before it reaches the application, and recovering a valid copy from storage when possible.

The final problem we consider in this work is the problem of faults in software, better known as **software bugs**. Research has shown that the number of software bugs is correlated with the number of lines of software source code [111], and that

even well-written software can be expected to have a nonnegligible number of bugs [57]. The storage stack is typically a large and complex part of the operating system, and thus, its implementation is likely to have many bugs. These bugs have the potential to subvert the correct operation of the storage stack. Since software bugs can change the software's behavior in arbitrary ways, it is possible to detect and recover from the results of certain classes of software bugs only. For example, detection is impossible if a software bug causes a *semantic failure*: behavior that is unintended but seemingly correct. Recovery is impossible if a software bug causes a *permanent* failure, which blocks overall progress by triggering repeatedly, or if the software bug causes damage beyond the scope of the recovery system. However, at the very least, the storage stack should be able to continue correct operation in the presence of software bugs that lead to detectable, *transient*, and sufficiently bounded forms of misbehavior.

1.2 Research approach and questions

Earlier, we stated that there is a spectrum of possible reliability improvements depending on how much reliability gain is desired versus how much cost one is willing to incur. In this dissertation, we are therefore required to make a number of choices. In this section, we first describe the choices that we consider as a given in the rest of our work, and then state the resulting research questions that we aim to answer.

First, as mentioned earlier, we focus on the software layer only. As such, we do not impose substantial restrictions on the underlying hardware, thus allowing us to employ our resulting solutions even on existing and low-cost hardware–that is, hardware that has not been especially engineered for reliability. Computer systems with such hardware stand to gain the most from reliability improvements in software. We will however also show that more recent CPU extensions can optionally be used to further reduce performance overheads.

Second, throughout our work, we explore the use of run-time *modularity* and *isolation* of the various software components in the storage stack. The use of isolated modules allows us to consider recovery from software failures and memory corruption on a per-component basis. Moreover, it allows the system to recover from a larger set of forms of misbehavior than possible on systems without such a strict separation between components. Thus, part of our research consists of exploring a meaningful definition and arrangement of components and boundaries within the storage stack. It is our point of view that both this exploration itself and the better guarantees justify the extra performance cost that must be paid for such fine-grained modularity and isolation. At the same time, we do not exclude the possibility that reduced versions of our solutions could also be applied in environments without such strict forms of separation.

With these two points as a given, our goal is to explore the end of the spectrum that optimizes for cost. In particular, we aim to maintain high performance,

acceptable resource usage, and low complexity, at the cost of legacy support, generality, and maximum reliability. In particular, we follow a number of main principles throughout our work.

Legacy: We explicitly refrain from limiting ourselves to solutions that integrate well with existing software systems. While we concede that legacy support generally allows for easier real-world adoption of any solutions, we believe that more significant benefits can be obtained by more structurally rethinking the system. As indicated previously, part of our work addresses problems that were in fact introduced as a direct result of the desire for legacy support.

Language: In order to maximize performance, we consider a storage stack that is written in the (unsafe) C language. We opt not to use safer but slower programming languages and paradigms. While these could decrease the possibility of low-level software bugs, operating system storage stacks are typically written in C for fast low-level access to data structures. We note that safer programming languages do not eliminate many classes of software bugs, nor do they deal with other problems such as soft memory errors any better than their unsafe counterparts.

Generality: We aim to exploit as much high-level knowledge as we can about the storage stack that is the target of our work. Thus, we intentionally sacrifice generality of our solutions. This allows us to optimize for both high performance and low complexity. For example, we can avoid handling cases that we know will never occur. As a result, a substantial part of our research attempts to explore how we can get the most out of our storage stack. While this approach runs the risk of making our solutions specific to our storage stack, we challenge some of our own assumptions in the last stage of our work.

Integration: In our research, we go beyond focusing on individual components and problems alone, and look at the "big picture" of generally improving the reliability of the entire storage stack. This allows us to study the effects of integrating solutions for individual components and problems. One aspect of such integration is that it allows us to reduce complexity further by exploring reuse of the same concepts and code for multiple purposes.

Transparency: We choose not to involve applications as active participants in our reliability solutions. Instead, whenever possible, the operating system should recover from reliability problems without exposing what happens to the applications in any way. The result is that applications need not be changed to benefit fully from our improvements. Application-transparent recovery is generally possible only in the case that the failure is transient–upon repeated or permanent failure, it may be impossible for the operating system to deliver certain services.

With the stated points in mind, this thesis focuses on the following main research questions:

- *Feasibility and properties*: It is possible to integrate a number of low-cost reliability techniques in order to construct a low-cost, high-gain reliability solution, and what properties does such a solution have?

- *Building blocks for reliability*: Is it beneficial to consider multiple types of reliability threats at once, rather than focusing on a single reliability threat for each solution; what common patterns and building blocks emerge if we do?

- *Exploiting high-level knowledge*: What benefits, if any, can be gained from exploiting high-level, specific knowledge about the storage stack for the purpose of improving reliability?

- *The ideal level of componentization*: What level of componentization of a storage stack into individual failure domains is appropriate from a reliability viewpoint, how should the components be arranged in order to provide optimal reliability guarantees, and how far can we push these guarantees while retaining low overheads?

- *Robustness against software bugs*: What kind of general model can be defined to characterize the solutions we develop to recover from software bugs in the various components of the storage stack, and how do our various solutions compare when examined using this model?

The last two research questions pertain primarily to the problem of software bugs. We examine this problem carefully in our work, since the problem is relatively unexplored in the context of storage stacks. In addition, we will later demonstrate that the solutions for software bugs also offer benefits in dealing with memory corruption.

Finally, we do not attempt to cover all cases of reliability problems exhaustively, even where theoretically possible. There is a general "long tail" of failures and especially combinations of failures that are increasingly unlikely to occur in practice. Covering all of those would require a large amount of complex and hard-to-test software additions.

1.3 Research overview

In this section, we present an overview of the research presented in this thesis. We describe Loris, our new storage stack arrangement (Sec. 1.3.1). We then describe the development platform and its advantages for reliability (Sec. 1.3.2). Finally, we summarize our solutions for the four aforementioned reliability problems (Sec. 1.3.3).

1.3.1 Loris: a new storage stack arrangement

The conceptual structure of the traditional storage stack, as can be found in most operating systems today, is shown in Fig. 1.1a. Its operation can be described in the following (simplified) way. Applications make file system calls into the Virtual File System (VFS) layer of the operating system, which forwards these calls to the

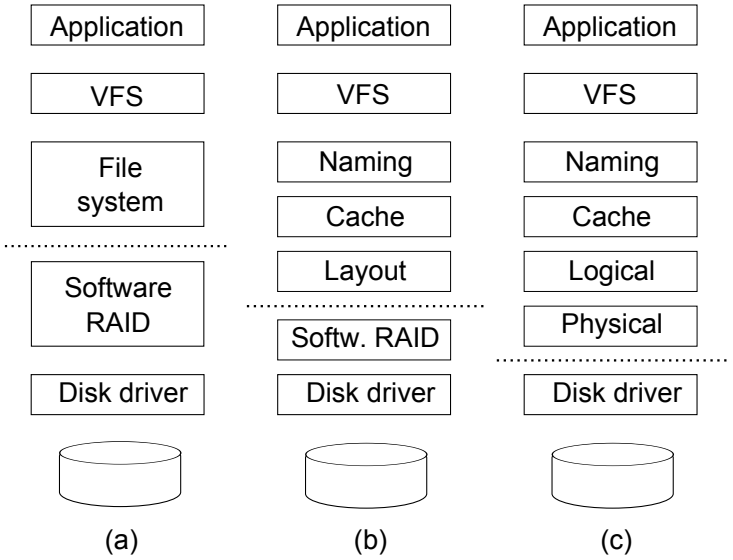


Figure 1.1: The figure depicts (a) the arrangement of layers in the traditional stack, (b) the first step in the rearrangement, and (c) the arrangement of the Loris storage stack. The layers above the dotted line are file-aware, the layers below are not.

appropriate file system. The file system converts the calls to *block operations*: requests to read or write certain disk blocks. Typically, these block operations are given directly to the storage device driver, which forwards them to the actual storage hardware. However, between the file system and driver layers, a software-RAID layer may add redundancy across multiple devices to protect against fail-stop device failures, without the file system being aware of this.

As we mentioned earlier, the traditional storage stack has fundamental reliability issues as a result of the *information gap* between the file system and the RAID layer. In addition, we will show that it has other shortcomings in the areas of heterogeneity and flexibility. Moreover, it lacks the level of modularity required for our work. Therefore, the first part of our work consists of rearranging the traditional storage stack to form a new storage stack which does not suffer from any of these issues.

To this end, we first split the traditional file system into three layers: a *naming* layer, which manages high-level abstractions such as directories and file attributes; a *cache* layer, which performs caching of file data; and, a *layout* layer, which converts file operations to block operations, and manages the layout on the underlying device. The result of this intermediate step is shown in Fig. 1.1b. After this split-up, we swap the traditional RAID layer with the layout layer, thereby making the former file-aware. We refer to the new file-based RAID layer as the *logical* layer, and to the layout layer as the *physical* layer, respectively. The physical layer consists of one or more instances (called *physical modules* or *file stores*), each independently responsible for one storage device. The four layers communicate with each other

in terms of **files**: storage containers that each have a unique identifier, byte-granular data contents of arbitrary size, and a small set of associated attributes.

The final result is shown in Fig. 1.1c. We call this arrangement the **Loris** storage stack. As we will demonstrate, Loris is a major step towards solving the aforementioned reliability, heterogeneity, flexibility, and modularity issues of the traditional stack.

Viewed from another perspective, the lower three new layers (cache, logical, physical) can together be seen as an implementation of **object storage** in the definition used in the distributed file systems world [38, 46]. This is shown in Fig. 1.2a and 1.2b. As a result, we also refer to files within the storage stack as **objects**. On top of the object store, the naming layer forms a namespace using the independent objects. For this reason, we also refer to the naming layer as the *namespace layer*.

There is largely a one-to-one relationship between files as exposed to applications and objects within the storage stack. However, each of the layers may need to use and store *metadata*: information necessary for the operation of the storage stack, but not exposed (directly) to the application layer. While the physical modules can store their metadata directly in designated areas on the storage device, the higher layers use the object abstraction to store their metadata in *metadata objects*. For example, the naming layer stores directories as objects.

1.3.2 The platform

As a proof of concept, we implement a prototype of Loris. This prototype serves as the basis of all our subsequent work. The implementation is developed on the MINIX 3 operating system [142]. MINIX 3 is based on the microkernel model, with a small kernel running with full system privileges (in kernel mode), and the rest of the operating system running in *system processes*: isolated processes running with normal privileges (in user mode) and in their own address space, with the ability to communicate with the rest of the system by means of interprocess communication (IPC).

For the purpose of improving system reliability, in particular in the light of software bugs, MINIX 3 has two important properties. First, the decomposition of the operating system into a number of system processes aids in both containment and (early) detection of the results of software bugs within each of these processes [62, 64]. The privileges of the system processes can be restricted at the IPC boundaries to the minimum needed for correct operation (the *principle of least authority* [122]), thereby restricting the damage that can be done as the result of anomalous behavior. Any system process's violation of its strict boundaries is sign of misbehavior and provides reason for its termination. The modularity allows MINIX 3 to provide additional facilities in this respect, such as a heart-beat check whether a system process is still responding. Throughout this work we refer to all forms of a system process visibly deviating from its intended behavior, and subsequently being terminated, as **process crashes**.

Second, MINIX 3 provides a basic recovery infrastructure for crashed system processes [63]. When a system process crashes, MINIX 3 has the ability to restart it automatically. However, the restarted instance of the system process will be started anew, and thus not retain any of its internal **state** from before the crash. In addition, ongoing IPC will be canceled. Thus, the system process and its communication partners are responsible for figuring out how to recover the previous state and resume IPC, if necessary. This basic infrastructure is sufficient to recover from crashes in several classes of device drivers [63], including storage (block) device drivers, since these have very little state to restore, and restarting IPC is simple as all operations are idempotent. Throughout this work, we use the term "stateless" for a system process with no state that, once lost due to a crash, needs to be recovered in order for the system process to resume correct operation. As we will show, most parts of the storage stack are *not* stateless.

1.3.3 Improving the reliability of Loris

The rearrangement of the storage stack already solves one major problem regarding *storage device failures*: by swapping the original file system layout layer and the original software-RAID layer, we close the information gap between these two layers. In particular, the swap ensures that no data will ever be propagated between devices (by the logical layer) without being checked for corruption (by the physical layer) first. However, the physical layer then needs a reliable way to detect data corruption. Our solution is to mandate that each physical module in the Loris stack use a form of checksumming called *parental checksumming*, which is sufficient to detect all forms of silent data corruption described earlier [88]. One layer higher, in the logical layer, redundancy across devices (and thus physical modules) is then used for recovery. The operation on a per-object basis of the logical layer allows both redundancy policies and handling of recovery failures to be established on a more fine-grained basis than in the traditional stack.

The new storage stack arrangement does complicate recovery from *whole-system failures*. In order for the storage stack to recover to an overall consistent state, an internally consistent snapshot of all metadata has to be reloaded from storage. In addition, any redundantly stored object data must be synchronized between the involved storage devices. Therefore, proper whole-system failure recovery involves the metadata of all layers and data stored across all the devices. Thus, while traditionally, consistency of metadata can be managed on a per-filesystem basis and consistency of data can be managed independently by the RAID layer, Loris must synchronize the metadata and replicated data across all layers and physical modules. Our solution consists of a combination of a stack-wide synchronization procedure to create new consistent *checkpoints* (or *recovery points*) for metadata, and a protocol between the logical and physical layers to both restore the last checkpoint and resynchronize any data outside the checkpoint.

For *memory corruption*, we focus on the Loris cache layer, as its cached object

data pages typically make up by far the largest part of the memory used by the storage stack and indeed the entire operating system. Our goal is to ensure that any memory corruption in these pages is at least detected, with high probability, before it propagates to the higher layers–in particular the application layer. However, since the cache is in the critical path of the majority of I/O operations, it is crucial that the overhead of such detection be kept low. Our solution consists of involving the cache layer in the process of generating checksums for data pages, which was previously implemented entirely in the physical layer. Once the cache layer is aware of checksums, it can use them for detection of memory corruption at little extra cost.

Finally, for *software bugs*, there is no single low-cost approach that provides a solution across the entire storage stack. Therefore, we look at the individual layers of the storage stack instead. As mentioned, the MINIX 3 crash recovery infrastructure already provides support that is sufficient for the recovery of (block) device drivers. Therefore, in this work, we consider recovering from device driver crashes to be a solved problem, and we focus on the other layers of the storage stack. All the modules in these layers have internal state which, in order to allow for application-transparent recovery, needs to be restored after a crash.

For the four core layers of the new arrangement (the naming, cache, logical, and physical layers), we come up with three separate solutions for process crash recovery, each with slightly different assumptions and recovery guarantees. For the logical and physical layers, we reuse part of the whole-system failure recovery infrastructure, combining it with an in-memory log that contains a recent history of operations in the cache layer. Thus, in these two layers, we integrate recovery from whole-system failures and process crashes. For recovery from crashes in the cache layer itself, keeping a copy of the state needed for recovery (dirty pages, among other things) would be too expensive, so we opt for recovery of this state from the memory image of the crashed process. In order to verify that the state has not been corrupted as part of the crash, we leverage the checksum support in the cache, thus integrating memory corruption detection and crash recovery support in this layer. For naming-layer crash recovery, we introduce support for transactions in the cache layer so as to make the naming layer stateless, thus allowing for straightforward process crash recovery.

This leaves the VFS layer. As it turns out, in particular this layer has more (numerous and diverse) internal state than can realistically be recovered from either itself or elsewhere. We therefore take a different approach for VFS. In several use cases, many applications running on the system will not actually need to interact with each other. Thus, we can place each application or group of interacting applications in its own isolated environment, and include a copy of parts of the operating system in this environment. In our case, each environment includes at least a copy of VFS. As a result, if an instance of VFS crashes, it will merely result in shutdown of its own environment, leaving the other environments unaffected. This approach requires a split of the operating system between the isolated environments (“domains”) and the shared base system (“host”). We propose a split between the namespace and object

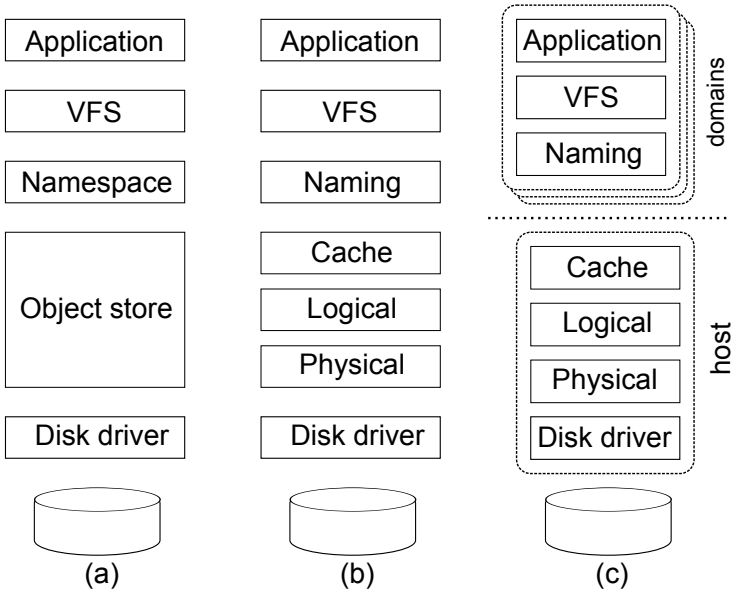


Figure 1.2: The figure depicts (a) the general model of an object storage stack, (b) Loris as an instance of such a stack, and (c) the split-up chosen for our virtualization approach: the layers above the dotted line are part of each of the virtual domains, whereas the layers below the dotted line are part of the shared base system.

storage layers, as shown in Fig. 1.2c. Taken to its logical conclusion, the result is a new approach to *virtualization*, with a number of interesting properties compared to the alternatives in this space: virtual machines [49] and operating system containers [131]. We present the idea and an initial implementation of its core in this thesis.

1.4 Contributions and thesis outline

We now present the organization of the rest of this thesis, and list the main contributions of each chapter.

In **Chapter 2**, we show that the traditional storage stack suffers from no less than six major shortcomings, in the areas of reliability, heterogeneity, and flexibility. We then introduce the Loris storage stack arrangement, and show how this new arrangement resolves all six issues. In terms of reliability, the main contribution of this chapter is a comprehensive solution for detection of storage device failures, consisting of the combination of a system of parental checksumming in the physical layers and the object awareness of the software RAID replacement (the logical layer). Chapter 2 is published as a technical report [10], which consists of the original paper published in Proceedings of the Sixteenth IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'10) [9] and an extended evaluation. The work in this chapter was done in collaboration with Raja Appuswamy, who au-

thored the idea; the design, implementation, and evaluation were done together. The same chapter appears in Raja Appuswamy's doctoral thesis [7], which focuses in particular on the flexibility side of Loris.

Chapter 3 focuses on whole-system failures, as well as process crashes in the logical and physical layers. We demonstrate that recovery from both failure types can be built on top of a unified infrastructure for creating and restoring checkpoints for metadata. On top of such a checkpointing framework, recovery from whole-system failures then only requires a system for data resynchronization, and recovery from process crashes in the lower Loris layers only requires a way to roll forward from the last checkpoint to the current state. We design, implement, and evaluate a checkpoint infrastructure, a data resynchronization protocol, and an in-memory roll-forward operations log in the cache layer. We define a set of requirements for physical-module implementations to support the checkpointing system, and we show how our approach to data resynchronization is inherently less costly than its traditional counterpart in RAID solutions. As proof of concept, we develop a new on-device layout and corresponding physical module which can create and restore checkpoints and perform resynchronization in a simple and reliable manner, called *TwinFS*. Chapter 3 was published in Proceedings of the Seventh IEEE International Conference on Networking, Architecture, and Storage (NAS'12) [150].

In **Chapter 4**, we focus on the cache layer. We consider the problems of detecting memory corruption in cached data and of recovering the cache-layer process after a crash. We argue that as part of the solution for both problems, the cache layer benefits from being involved in the checksumming of data, which was previously limited to the physical layer. Within the cache layer, we show that these data checksums can be used for two purposes. First, the checksums can be used for detection of memory corruption in cached data. We demonstrate that simple algorithms can achieve a high probability of detection of corruption at low cost. Second, in the event of a crash in the cache layer process, the checksums can be used to verify that dirty data pages have not been corrupted as part of the crash, so that a new instance of the cached process can recover them from the old instance. We show that these data checksums can be maintained in a checksum-protected in-memory metadata hierarchy at low cost. Finally, we show that combining the two solutions yields advantages for both. Chapter 4 was published in Proceedings of the Sixth Latin-American Symposium on Dependable Computing (LADC'13) [151].

In **Chapter 5**, we consider process crashes in the naming layer. We start by arguing that this layer may in fact contain not only the traditional (*primary*) namespace manager of the storage stack, but also additional (*extension*) namespace modules which may expose the contents of a single, internally hierarchically structured object through the normal file system application programming interface. We then argue that both these categories of namespace modules may experience crashes, albeit for different reasons. As a first step towards crash recovery, we argue that the modules should be stateless, and thus not defer sending any modifications to the cache layer. In addition, to prevent inconsistencies resulting from crashes in the middle of

processing a system call, we extend the cache layer with transaction support. Each namespace module can use these transactions to make atomic transitions from one consistent state to another. Finally, we formulate a number of additional rules which guarantee successful namespace module crash recovery. We evaluate our ideas using both the Loris primary namespace module, and a newly developed HDF5 [58] extension namespace module. Chapter 5 was published in Proceedings of the Nineteenth IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'13) [152].

Chapter 6 presents our new virtualization concept, originally conceived as a way to create smaller failure domains to deal with crashes in the VFS process, extended into a viable alternative to other virtualization technologies. The heart of the approach consists of a new split of the storage stack, between the lower layers that form the object store (the cache layer and below) and the upper layers that support application abstractions (the naming layer and above). This allows both isolation at the application level and sharing at the object level. We describe the conceptual advantages and disadvantages of the approach, and present an initial prototype implementation. Chapter 6 was published in Proceedings of the Seventh ACM International Systems and Storage Conference (SYSTOR'14) [154].

In **Chapter 7**, we take all the individual reliability and virtualization extensions presented in the previous chapters and combine them to form a reliable, virtualizing version of the Loris storage stack. This final integration allows us to look at the bigger picture of storage stack reliability, and to answer a number of research questions that exceed the scope of each of the individual projects. We perform the actual work in two steps. First, we extend Loris to include a complete version of the storage virtualization features as laid out in Chapter 6: object virtualization, copy-on-write, and deduplication. We call the result *vLoris*. Second, we integrate the reliability extensions from Chapters 3 to 5, making them work with both the virtualization extensions and each other, thereby forming *rvLoris*. We evaluate the advantages, limitations, complexity, performance, and reliability of the resulting storage stack. Chapter 7 was published in Proceedings of the Second International Symposium on Computing and Networking (CANDAR'14) [153].

Chapters 2 to 7 have been reproduced here exactly as they were published, with the exception of a handful of corrected typos. **Chapter 8** elaborates on a number of specific points which we did not manage to fit in the original publications.

In **Chapter 9**, we conclude the thesis, by summarizing our work, returning to our originally stated research questions, and listing possible areas of future work.

2

Loris - A Dependable, Modular, File-Based Storage Stack

Abstract

The arrangement of file systems and volume management/RAID systems, together commonly referred to as the storage stack, has remained the same for several decades, despite significant changes in hardware, software and usage scenarios. In this paper, we evaluate the traditional storage stack along three dimensions: reliability, heterogeneity and flexibility. We highlight several major problems with the traditional stack. We then present Loris, our redesign of the storage stack, and we evaluate several reliability, availability and performance aspects of Loris.

2.1 Introduction

Over the past several years, the storage hardware landscape has changed dramatically. A significant drop in the cost per gigabyte of disk drives has made techniques that require a full disk scan, like *fsck* or whole disk backup, prohibitively expensive. Large scale installations handling petabytes of data are very common today, and devising techniques to simplify management has become a key priority in both academia and industry. Research has revealed great insights into the reliability characteristics of modern disk drives. "Fail-partial" failure modes [116] have been studied extensively and end-to-end integrity assurance is more important now than ever before. The introduction of SSDs and other flash devices is sure to bring about a sea change in the storage subsystem. Radically different price/performance/reliability trade-offs have necessitated rethinking several aspects of file and storage management [42, 74]. In short, the storage world is rapidly changing and our approach to making storage dependable must change, too.

Traditional file systems were written and optimized for block-oriented hard disk drives. With the advent of RAID techniques [113], storage vendors started developing high-performance, high-capacity RAID systems in both hardware and software. The block-level interface between the file system and disk drives provided a convenient, backward-compatible abstraction for integrating RAID algorithms.

As installations grew in size, administrators needed more flexibility in managing file systems and disk drives. Volume managers [144] were designed as block-level drivers to break the "one file system per disk" bond. By providing logical volumes, they abstracted out details of physical storage and thereby made it possible to resize file systems/volumes on the fly. Logical volumes also served as units of policy assignment and quota enforcement. Together, we refer to the RAID and volume management solutions as the *RAID layer* in this paper.

This arrangement of file system and RAID layers, as shown in Figure 2.1a, has been referred to as the storage stack [33]. Despite several changes in the hardware landscape, the traditional storage stack has remained the same for several decades. In this paper, we examine the block-level integration of RAID and volume management along three dimensions: reliability, flexibility, and heterogeneity. We highlight several major problems with the traditional stack along all three dimensions. We then present Loris, our new storage stack. Loris improves modularity by decomposing the traditional file system layer into several self-contained layers, as shown in Figure 2.1b. It improves reliability by integrating RAID algorithms at a different layer compared to the traditional stack. It supports heterogeneity by providing a file-based stack in which the interface between layers is a standardized file interface. It improves flexibility by automating device administration and enabling per-file policy selection.

This paper is structured as follows. In Sec. 2.2, we explain in detail the problems associated with the traditional stack. In Sec. 2.3, we briefly outline some attempts taken by others in redesigning the storage stack and also explain why other

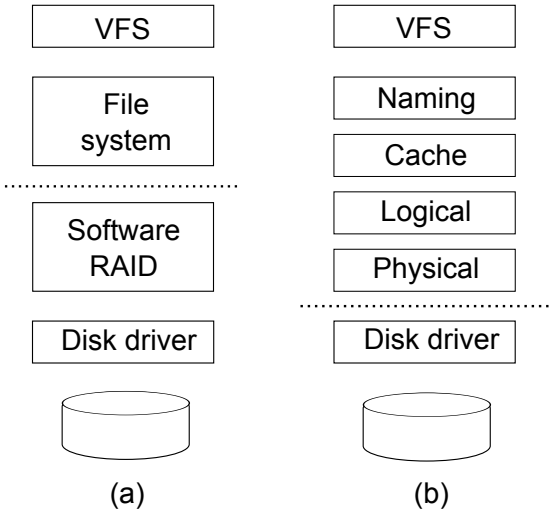


Figure 2.1: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file-aware, the layers below are not.

approaches fail to solve all the problems. In Sec. 2.4, we introduce Loris and explain the responsibilities and abstraction boundaries of each layer in the new stack. We also sketch the realization of these layers in our prototype implementation. In Sec. 2.5, we present the advantages of Loris. We then evaluate both performance and reliability aspects of Loris in Sec. 2.6 and conclude in Sec. 2.7.

2.2 Problems with the Traditional Storage Stack

In this section, we present some of the most important problems associated with the traditional storage stack. We present the problems along three dimensions: reliability, flexibility and heterogeneity.

2.2.1 Reliability

The first dimension is reliability. This includes the aspects of data corruption, system failure, and device failure.

Data Corruption

Modern disk drives are not "fail-stop." Recent research has analyzed several "fail-partial" failure modes of disk drives. For example, a *lost write* happens when a disk does not write a data block. A *misdirected* write by the disk results in a data block being written at a different position than its intended location. A *torn write* happens when a write operation fails to write all the sectors of a multisector data

block. In all these cases, if the disk controller (incorrectly) reports back success, data is silently corrupted. This is a serious threat to data integrity and significantly affects the reliability of the storage stack.

File systems and block-level storage systems detect data corruption by employing various checksumming techniques [128]. The level of reliability offered by a checksumming scheme depends heavily on what is checksummed and where the checksum is stored. Checksums can be computed on a per-sector or per-block (file system block) basis, where a block is typically 2, 4, 8, or more sectors. Using per-sector checksums does not protect one against any of the failure cases mentioned above if checksums are stored with the data itself. Block checksums, on the other hand, protect against torn writes but not against misdirected or lost writes.

In yet another type of checksumming, called *parental checksumming*, the checksum of a data block is stored with its parent. For instance, a parental checksumming implementation could store block checksums in the inode, right next to the block pointers. These checksums would then be read in with the inode and used for verification. Formal analysis has verified that parental checksumming detects all of the aforementioned sources of corruption [88].

However, using parental checksumming in the current storage stack increases the chance of data loss significantly [88]. Since the parental relationship between blocks is known only to the file system, parental checksums can be used only by the file system layer. Thus, while file system initiated reads can be verified, any reads initiated by the RAID layer (for partial writes, scrubbing, or recovery) escape verification. As a result, a corrupt data block will not only go undetected by the RAID layer, but could also be used for parity recomputation, causing parity pollution [88], and hence data loss.

System Failure

Crashes and power failures pose a *metadata consistency* problem for file systems. Several techniques like soft updates and journaling have been used to reliably update metadata in the face of such events. RAID algorithms also suffer from a *consistent update* problem. Since RAID algorithms write data to multiple disk drives, they must ensure that the data on different devices are updated in a consistent manner.

Most hardware RAID implementations use NVRAM to buffer writes until they are made durable, cleanly side-stepping this problem. Several software RAID solutions, on the other hand, resort to whole disk *resynchronization* after an unclean shutdown. During resynchronization, all data blocks are read in, parity is computed, and the computed parity is verified with the on-disk parity. If a mismatch is detected, the newly computed parity is written out replacing the on-disk parity.

This approach—in addition to becoming increasingly impractical due to the rapid increase in disk capacity—has two major problems: (1) it increases the vulnerability window within which a second failure can result in data loss. This problem is also known as the “RAID write hole”, and (2) it adversely affects availability, as

the whole disk array has to be offline during the resynchronization period [20].

The other approach adopted by some software RAID implementations is journaling a block bitmap to identify regions of activity during the failure. While this approach reduces resynchronization time, it has a negative impact on performance [34], and results in functionality being duplicated across multiple layers.

Device Failure

Storage array implementations protect against a fixed number of disk failures. For instance, a RAID 5 implementation protects against a single disk failure. When an unexpected number of failures occur, the storage array comes to a grinding halt. An ideal storage array however, should degrade gracefully. The amount of data inaccessible should be proportional to the number of failures in the system. Research has shown that to achieve such a property, a RAID implementation must provide: (1) selective replication of metadata to make sure that the directory hierarchy remains navigable at all times, and (2) fault-isolated positioning of files so that a failure of any single disk results in only files on that disk being inaccessible [129].

By recovering *files* rather than *blocks*, file-level RAID algorithms reduce the amount of data that must be recovered, thus shrinking the vulnerability window before a second failure. Even a second failure during recovery results in the loss of only some file(s), which can be restored from backup sources, compared to block-level RAID where the entire array must be restored. None of these functionalities can be provided by the traditional storage stack as the traditional RAID implementation operates strictly below a block interface.

2.2.2 Flexibility

We discuss two points pertaining to flexibility: management and policy assignment.

Management Flexibility

While traditional volume managers make device management and space allocation more flexible, they introduce a series of complex, error prone administrative operations, most of which should be automated. For instance, a simple task such as adding a new disk to the system involves several steps like creating a physical volume, adding it to a volume group, expanding logical volumes and finally resizing file systems. While new models of administering devices, like the storage pool model [136], are a huge improvement, they still suffer from other problems, which we will describe in the next section.

In addition to device management complexity, software-based RAID solutions expose a set of tunable parameters for configuring a storage array based on the expected workload. It has been shown that an improperly configured array can render layout optimizations employed by a file system futile [134]. This is an example of the more general "information gap" problem [33]— different layers performing

different optimizations unaware of the effect they might have on the overall performance.

Policy Assignment Flexibility

Different files have different levels of importance and need different levels of protection. However, policies like the RAID level to use, encryption, and compression, are only available on a per-volume basis rather than on a per-file basis. Several RAID implementations even lock-in the RAID levels and make it impossible to migrate data between RAID levels. In cases where migration is supported, it usually comes at the expense of having to perform a full-array dump and restore. In our view, an ideal storage system should be flexible enough to support policy assignment on a per-file, per-directory, or a per-volume basis. It should support migration of data between RAID levels on-the-fly without affecting data availability.

2.2.3 Heterogeneity Issues

New devices are emerging with different data access granularities and storage interfaces. Integrating these devices into the storage stack has been done using two approaches that involve extending either the file system layer or the block-level RAID layer with new functionality.

The first approach involves building file systems that are aware of device-specific abstractions [74]. However, as the traditional block-based RAID layer exposes a block interface, it is incompatible with these file systems. As a result, RAID and volume management functionalities must be implemented from scratch for each device family.

The second approach is to be backward compatible with traditional file systems. This is typically done by adding a new layer that translates block requests from the file system to device-specific abstractions [42]. This layer cannot be integrated between the file system and RAID layers, as it is incompatible with the RAID layer. Hence, such a layer has to either reimplement RAID algorithms for the new device family, or be integrated below the RAID layer. This integration retains compatibility with both traditional file system and RAID implementations. However, this has the serious effect of widening the information gap by duplicating functionality. For instance, both the file system and translation layers now try to employ layout optimizations – something that is completely unwarranted. The only way to avoid this duplication is by completely redesigning the storage stack from scratch.

2.3 Solutions Proposed in the Literature

Several approaches have been taken to solving some of the problems mentioned in the previous section. However, none of these approaches solve all these problems by design. In this section, we highlight only the most important techniques. We

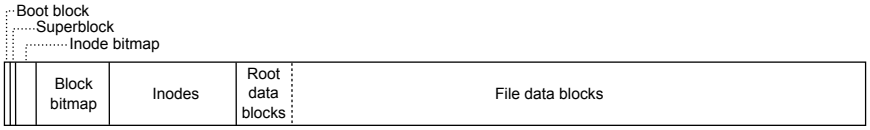


Figure 2.2: High-level overview of the on-disk layout used by the physical layer prototype.

classify the approaches taken into four types, namely: (1) inferring information, (2) sharing information, (3) refactoring the storage stack, and (4) extending the stack with stackable filing.

One could compensate for the lack of information in the RAID layer by having it infer information about the file system layer. For instance, semantically smart disks [129] infer file system specific information (block typing and structure information), and use the semantic knowledge to improve RAID flexibility, availability, and performance. However, by their very nature, they are file-system-specific, making them nonportable.

Instead of inferring information, one could redesign the interface between the file system and RAID layers to share information. For example, ExRAID [33], a software RAID implementation, provides array related information (such as disk boundaries and transient performance characteristics of each device) to an informed file system (I-LFS), which uses it to make informed data placement decisions.

While both inferring and sharing information can be used to add new functionality, they do not change the fundamental division of labor between layers. Hence, most of the problems we mentioned remain unresolved.

A few projects have refactored the traditional storage stack. For instance, ZFS [136]'s storage stack consists of three layers, the ZFS Posix Layer (ZPL), the Data Management Unit (DMU), and the Storage Pool Allocator (SPA). ZFS solves the reliability and flexiblity problems we mentioned earlier by merging block allocation with RAID algorithms in its SPA layer. SPA exposes a virtual block abstraction to DMU and acts as a multidisk block allocator. However, because SPA exposes a block interface, it suffers from the same heterogeneity problems as the RAID layer. In addition, we believe that layout management and RAID are two distinct functionalities that should be modularized in separate layers.

RAIF [75] provides RAID algorithms as a stackable file system. RAID algorithms in RAIF work on a per-file basis. As it is stackable, it is very modular and can be layered on any file system, making it device independent. While this does solve the flexibility and heterogeneity problems, it does not solve the reliability problems.

2.4 The Design of Loris

We now present our new storage stack, Loris. The Loris stack consists of four layers in between the VFS layer and the disk driver layer, as shown in Figure 2.1b.

Within the stack, the primary abstraction is the *file*. Each layer offers an interface for creating and manipulating files to the layer above it, exposing per-file operations such as *create*, *read*, *write*, *truncate*, and *delete*. A generic *attribute* abstraction is used to both maintain per-file metadata, and exchange information within the stack. The *getattr* and *setattr* operations retrieve and set attributes. Files and attributes are stored by the lowest layer.

We implemented a Loris prototype on the MINIX 3 multiserver operating system [62]. The modular and fault-tolerant structure of MINIX 3 allows us to quickly and easily experiment with invasive changes. Moreover, we plan to apply ongoing research in the areas of live updates [47] and many-core support to our work. MINIX 3 already provides VFS and the disk drivers, each running as a separate process, which improves dependability by allowing failed OS components to be replaced on the fly.

The four layer implementations of the prototype can be combined into one process, or separated out into individual processes. The single-process setup allows for higher performance due to fewer context switches and less memory copying overhead. The multiprocess setup by nature imposes a very strict separation between layers, provides better process fault isolation in line with MINIX 3's dependability objectives [62], and is more live-update-friendly [47]. The difference between these configurations is abstracted away by a common library that provides primitives for communication and data exchange.

We will now discuss each of the layers in turn, starting from the lowest layer.

2.4.1 The Physical Layer

The lowest layer in the Loris stack is the *physical layer*. The physical layer algorithms provide device-specific layout schemes to store files and attributes. They offer a *fail-stop physical file* abstraction to the layers above it. By working on physical files, the rest of the layers are isolated from device-specific characteristics of the underlying devices (such as access granularity).

As the physical files are fail-stop, every call that requests file data or attributes, returns either a result that has been verified to be free of corruption or an error. To this end, every physical layer algorithm is required to implement parental checksumming. To repeat, above the physical layer, there is no silent corruption of data. A torn, lost, or misdirected write is converted into a hard failure that is passed upward in the stack.

In our prototype, each physical device is managed by a separate, independent instance of one of the physical layer algorithms. We call such an instance a *module*. Each physical module has a global *module identifier*, which it uses to register to the logical layer at startup.

On-Disk Layout

The on-disk layout of our prototype is based directly on the MINIX 3 File System (MFS) [142], a traditional UNIX-like file system. We have extended it to support parental checksums. Figure 2.2 shows a high-level view of the layout. We deliberately chose to stay close to the original MFS implementation so that we could measure the overhead incurred by parental checksumming.

Each physical file is represented on disk by an *inode*. Each inode has an *inode number* that identifies the physical file. The inode contains space to store persistent attributes, as well as 7 direct, one single indirect and one double indirect *safe block pointers*. Each safe block pointer contains a block number and a CRC32 checksum of the block it points to. The single and double indirect blocks store such pointers as well, to data blocks and single indirect blocks, respectively. All file data are therefore protected directly or indirectly by checksums in the file inode.

The inodes and other metadata are protected by means of three special on-disk files. These are the inode bitmap file, the block bitmap file, and the "root file." The bitmap file inodes and their indirect blocks contain safe block pointers to the bitmap blocks. The root file forms the hierarchical parent of all the inodes—its data blocks contain checksums over all inodes, including the bitmap file inodes. The checksums in the root file are stored and computed on a per-inode basis, rather than a per-block basis. The checksum of the root file's inode is stored in the superblock.

Figure 2.3 shows the resulting parental checksumming hierarchy. The superblock and root data blocks contain only checksums; the inodes and indirect blocks contain safe block pointers. Please note that this hierarchy is used only for checksumming—unlike copy-on-write layout schemes [66], blocks are updated in-place.

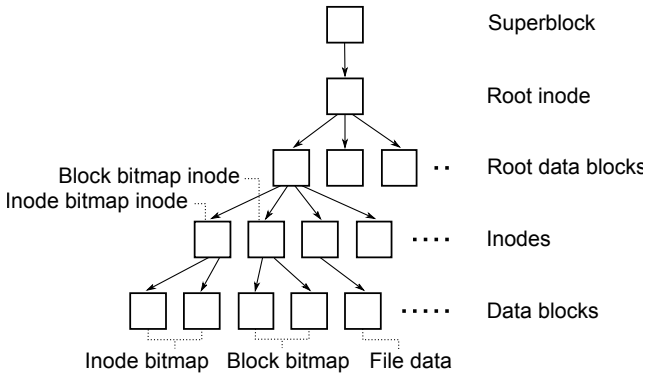


Figure 2.3: Parental checksumming hierarchy used by the physical layer prototype. With respect to parental checksumming, the two special bitmap files are treated as any other files. Indirect blocks have been omitted in this figure.

Delayed Checksum Computation

One of the main drawbacks of parental checksumming is cascading writes. Due to the inherent nature of parental checksumming, a single update to a block could result in several blocks being updated all the way up the parental tree to the root. Updating all checksums in the mainstream write path would slow down performance significantly.

The physical layer prevents this by delaying the checksum computation, using a small metadata block cache that is type-aware. By knowing the type of each block, this cache makes it possible to perform checksum computation only when a block is written to or read from the underlying device. For instance, by knowing whether a block is a bitmap or inode block, it can compute and update checksums in the bitmap and root file inodes lazily, that is, only right before flushing these metadata blocks from the cache.

Error Handling

Parental checksumming allows the physical layer to detect corrupt data. When a physical module detects a checksum error in the data or indirect block of a file, it marks that portion of the file as corrupted. If the file's inode checksum, as stored in the root file, does not match the checksum computed over the actual inode, then the entire file is marked as corrupted. In both cases, reads from the corrupt portion will result in a checksum error being returned to the logical layer. The physical module uses two attributes in the file's inode, begin range and end range, to remember this *sick range*.

For instance, consider a read request to a physical module for the first data block of a file. If the physical module detects a checksum mismatch on reading in the data block, it sets the begin and end range attributes to 0 and 4095 respectively, and responds back to the logical layer with a checksum error. We will detail the recovery process initiated by the logical layer when we describe its error handling mechanism.

In contrast to inodes and data blocks, if a part of the other on-disk metadata structures is found corrupted, the physical module will shut down for offline repair. While we could have supported metadata replication to improve reliability, we chose not to do so for the first prototype to stay as close as possible to the original MFS implementation in order to get honest measurements.

If the underlying device driver returns an error or times out, the physical module will retry the operation a number of times. Upon repeated failure, it returns back an I/O error to the logical layer. An I/O error from the physical module is an indication to the logical layer that a fatal failure has occurred and that the erroneous device should not be used anymore.

2.4.2 The Logical Layer

The logical layer implements RAID algorithms on a per-file basis to provide various levels of redundancy. The logical layer offers a *reliable logical file* abstraction to the layers above. It masks errors from the physical layer whenever there is enough data redundancy to recover from them. One logical file may be made up of several independent physical files, typically each on a different physical module, and possibly at different locations. As such, the logical layer acts as a centralized multiplexer over the individual modules in the physical layer.

Our prototype implements the equivalents of RAID levels 0, 1, and 4—all of these operate on a per-file basis. There is only one module instance of the logical layer, which operates across all physical modules.

File Mapping

The central data structure in our logical layer prototype is the *mapping*. The mapping contains an entry for every logical file, translating *logical file identifiers* to *logical configurations*. The logical configuration of a file consists of (1) the file's RAID level, (2) the stripe size used for the file (if applicable), and (3) a set of one or more physical files that make up this logical file, each specified as a physical module and inode number pair. The RAID level implementations decide how the physical files are used to make up the logical file.

The *create* operation creates a mapping entry for a given logical file identifier, with a given configuration (more about this later). For all other file operations coming in from above, the logical layer first looks up the file's logical configuration in the mapping. The corresponding RAID algorithm is then responsible for calling appropriate operations on physical modules.

Figure 2.4 shows how a logical file that is striped across two devices, is constructed out of two independent physical files. The mapping entry for this file F1 could look like this: F1=<raidlevel=0, stripesize=4096, physicalfiles=<D1:I1,

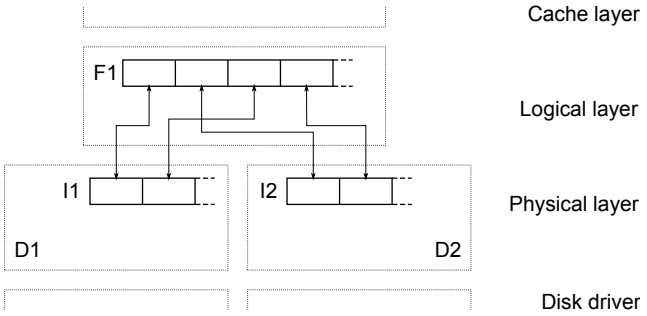


Figure 2.4: An example of the file abstractions provided by the logical and physical layers. The logical layer exposes a logical file, F1, which is constructed out of two physical files, namely I1 on physical module D1 and I2 on physical module D2, by means of striping.

D2:I2>>. The entry specifies a RAID level of 0, a stripe size of 4096 bytes, and two physical files: file I1 on physical module D1 and file I2 on physical module D2. Now consider a read request for the first 16384 bytes of this file coming down to the logical layer. Upon receiving the read request, the logical layer looks up the entry for F1 in its mapping, and passes on the call to the RAID 0 algorithm. The RAID 0 code uses the entry to determine that logical bytes 0-4095 are stored as bytes 0-4095 in physical file D1:I1, logical bytes 4096-8191 are stored as bytes 0-4095 in file D2:I2, logical bytes 8192-12287 are stored as bytes 4096-8191 in file D1:I1, and logical bytes 12288-16383 are stored as bytes 4096-8191 in file D2:I2. The logical layer issues two read requests, one to D1 for the first 8192 bytes of I1, and the other to D2 for the first 8192 bytes of I2. The results are combined to form a "flat" result for the layer above.

The mapping itself is a logical file. The logical configuration of this file is hard-coded. The mapping file is crucial for accessing any other files, and is therefore mirrored across all physical modules for increased dependability. It uses the same static inode number on all physical modules.

Error Handling

When the logical layer gets a checksum error in response to an operation on a physical module, it will restore the correct data for the file involved in the operation if enough data redundancy is present. If on-the-fly restoration is not possible, the logical layer fails the operation with an I/O error.

For instance, let us consider a read request for the first block of a file mirrored on two physical modules P1 and P2. If P1 responds back with a checksum error, the logical layer first retrieves the begin and end sick range attributes from P1. Assuming that only the first block was corrupt, these values would be 0 and 4095. The logical layer then issues a read request to module P2, for data in the range 0–4095. If this read request succeeds, the logical layer issues a write request to P1 for this data range, thereby performing on-the-fly recovery. It finally clears the sick range by resetting begin and end ranges to their defaults.

When the logical layer gets an I/O error from a physical module, it considers this to be a permanent error, and disables the physical module. The logical layer will continue to serve requests for affected files from redundant copies where possible, and return I/O errors otherwise.

2.4.3 The Cache Layer

The cache layer caches file data in main memory. This layer may be omitted in operating systems that provide a unified page cache. As MINIX 3 does not have a unified page cache, our prototype implements this layer. The cache layer is also needed on systems that do not have any local physical storage such as PDAs and other small mobile devices.

File-Based Caching

The prototype cache is file-aware and performs readahead and eviction of file data on a per-file basis. Files are read from the lower layers in large readahead chunks at a time, and only entire files are evicted from the cache. The cache maintains file data at the granularity of memory pages.

Early experiments showed a large performance penalty incurred by small file writes. Small file writes were absorbed completely by the cache until a sync request was received or the cache needed to free up pages for new requests. During eviction, each file would be written out by means of an individual *write* operation, forcing the physical layer to perform a large number of small random writes. To counter this problem, we introduced a *vwrite* call to supply a vector of write operations for several files. The cache uses this call to pass down as many dirty pages as possible at once, eventually allowing the physical modules to reorder and combine the small writes.

Problems with Delayed Allocation

The cache delays writes, so that *write* calls from above can be satisfied very quickly. This results in allocation of data blocks for these writes to be delayed until the moment that these blocks are flushed out. This delayed allocation poses a problem in the stack. Because of the abstraction provided by the logical layer, the cache has no knowledge about the devices used to store a file, nor about the free space available on those devices. Therefore, when a write operation comes in, the cache cannot determine whether the write will eventually succeed.

Although we have not yet implemented a solution for this in our prototype, the problem can be solved by means of a free-space reservation system exposed by the physical modules through the logical layer to the cache.

2.4.4 The Naming Layer

The naming layer is responsible for naming and organizing files. Different naming layer algorithms can implement different naming schemes: for example, a traditional POSIX style naming scheme, or a search-oriented naming scheme based on attributes.

POSIX Support

Our prototype implements a traditional POSIX naming scheme. It processes calls coming from the VFS layer above, converting POSIX operations into file operations.

Only the naming layer knows about the concept of directories. Below the naming layer, directories are stored as files. The naming layer itself treats directories as flat arrays of statically sized entries, one per file. Each entry is made up of a file name and a logical file identifier. Again, this layout was chosen for simplicity and to stay

close to the original MFS implementation for comparison purposes. A new naming module could implement more advanced directory indexing.

The naming layer is also responsible for maintaining the POSIX attributes of files (file size, file mode, link count, and so on). It uses Loris attributes for this: it uses the *setattribute* call to send down POSIX attribute changes, which are ultimately processed and stored by the physical layer in the file's inode.

Policy Assignment

When creating a new file, the naming layer is responsible for picking a new logical file identifier, and an initial logical configuration for the file. The logical configuration may be picked based on any information available to the naming layer: the new file's name, its containing directory, its file type, and possibly any flags passed in by the application creating the file. The chosen logical configuration is passed to lower layers in the *create* call in the form of attributes.

By default, directories are mirrored across all devices in order to provide graceful degradation. Upon getting a *create directory* request from VFS, the naming layer picks a new file identifier for the directory, and sends down a *create* call to the cache, with RAID level 1 specified as the file's logical policy. The cache forwards the call to the logical layer. The logical layer creates a new entry in the mapping for this file, and forwards the create call to all of the physical modules. Upon return, the logical layer stores the resulting inode numbers in the mapping entry as well.

2.5 The Advantages of Loris

In this section, we highlight how Loris solves all the problems mentioned in Sec. 2.2 by design. This section has been structured to mirror the structure of Sec. 2.2 so that readers can match the problems with their corresponding solutions one-to-one.

2.5.1 Reliability

We now explain how Loris protects against the three threats to data integrity.

Data Corruption

As RAID algorithms are positioned in the logical layer, all requests, both user application initiated reads and RAID initiated reads, are serviced by the physical layer. Thus, any data verification scheme needs to be implemented only once, in the physical layer, for all types of requests. In addition, since the physical layer is file-aware, parental checksumming can be used for detecting all possible sources of corruption. Thus, by requiring every physical layer algorithm to implement parental checksumming, fail-partial failures are converted into fail-stop failures. RAID algorithms can safely provide protection against fail-stop failures without propagating corruption.

System Failure

Journaling has been used by several file systems to provide atomic update of system metadata [53]. Modified journaling implementations have also been used to maintain the consistency between data blocks and checksums in the face of crashes [133]. While any such traditional crash recovery techniques can be used with Loris to maintain metadata consistency, we are working on a new technique called *metadata replay*. It protects the system from both hard crashes (power failures, kernel panic, etc.) and soft crashes (modules failing due to bugs). We will provide a brief description of this technique now, but it should be noted that independently of the technique used, Loris does not require expensive whole disk synchronization due to its file-aware nature.

Metadata replay is based on the counterintuitive idea that user data (file data and POSIX attributes), if updated atomically, can be used to regenerate system metadata. To implement this, we have two requirements: (1) a lightweight mechanism to log user data, and (2) some way to restore back the latest consistent snapshot. With highly flexible policy selection in place, the user could log only important files, reducing the overhead of data logging. We plan to add support for selective logging and metadata snapshotting. When a crash occurs, the logical layer coordinates the rollback of all physical layers to a globally consistent state. Then, the logged user data are replayed, regenerating metadata in both logical and physical layers, and bringing the system to new consistent state.

Device Failure

Graceful degradation is a natural extension of our design. Since RAID policies can be selected on a per-file basis, directories can be replicated on all devices while file data need not be, thereby providing selective metadata replication. Since the logical layer is file-aware, fault-isolated placement of files can also be provided on a per-file basis. Furthermore, recovery to a hot spare on a disk failure is faster than a traditional RAID array since the logical layer recovers files. As mentioned earlier, file-granular recovery restores only "live data" by nature, i.e., unused data blocks in all physical layers do not have to be restored. Because traditional RAID operates at the block level, it is unaware of which data blocks hold file data, and has to restore all data blocks in the array.

2.5.2 Flexibility

The new file-oriented storage stack is more flexible than the traditional stack in several ways. Loris supports automating several administrative tasks, simplifies device management, and supports policy assignment at the granularity of files.

Management Flexiblity

Loris simplifies administration by providing a simple model for both device and quota management. It supports automating most of the traditional administrative chores. For instance, when a new device is added to an existing installation, Loris automatically assigns the device a new identifier. A new physical module corresponding to this device type is started automatically and this module registers itself with the logical layer as a potential source of physical files. From here on, the logical layer is free to direct new file creation requests to the new module. It can also change the RAID policy of existing files on-the-fly or in the background. Thus, Loris supports a pooled storage model similar to ZFS.

File systems in ZFS [136] serve as units of quota enforcement. By decoupling volume management from device management, these systems make it possible for multiple volumes to share the same storage space. We are working on a file volume management implementation for Loris. We plan to modify the logical layer to add support for such a volume manager. File volumes in Loris will be able to provide functionalities similar to ZFS since files belonging to any volume can be allocated from any physical module.

Policy Assignment Flexiblity

Loris provides a clean split between policy and mechanism. For instance, while RAID algorithms are implemented in the logical layer, the policy that assigns RAID levels to files can be present in any layer. Thus, while the naming layer can assign RAID levels on a per-file, per-directory or even per-type basis [75], the logical layer could assign policies on a per-volume basis or even globally across all files.

2.5.3 Heterogeneity

All of the aforementioned functionalities are device-independent. By having the physical layer provide a physical file abstraction, algorithms above the physical layer are isolated from device-specific details. Thus, Loris can be used to set up an installation where disk drives coexist with byte-granular flash devices and Object-based Storage Devices (OSDs), and the administrator would use the same management primitives across these device types. In addition, as the physical layer works directly on the device without being virtualized, device-specific layout optimizations can be employed without creating an information gap.

2.6 Evaluation

In this section, we will evaluate several reliability, availability and performance aspects of our prototype.

2.6.1 Test Setup

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4 GB RAM, and four 500 GB 7200RPM Western Digital Caviar Blue SATA hard disks (WD5000AAKS), three of which were connected to separate ICIDU SATA PCI EXPRESS cards. We ran all tests on 8 GB test partitions at the beginning of the disks. In experiments where Loris is compared with MFS, both were set up to work with a 32 MB buffer cache.

2.6.2 Evaluating Reliability and Availability

To evaluate the reliability and availability of Loris, we implemented a fault injection block driver that is capable of simulating both fail-partial failures (by corrupting specific data blocks) and fail-stop disk failures (by returning an EIO on all requests).

We first present an evaluation of the ability of Loris to perform on-the-fly data recovery. Rather than just showing that our prototype detects corruption, we illustrate how file-awareness helps in reducing the recovery time. We then present an evaluation of availability under unexpected failures. We show two cases of graceful degradation, both of which cannot be done by block-level RAID implementations.

On-the-Fly Recovery

Our recovery measurements were gathered using a series of fault injection tests. The test file system consists of a single 100 MB file, mirrored over a two-disk Loris installation. The test workload is generated by a user application that issues read requests for specific data ranges in the file. These read requests get forwarded through the stack to the fault injection driver.

The driver corrupts three types of file blocks in three different scenarios: (1) a random direct data block, (2) a random single indirect block, and (3) the double indirect block. In all cases, the driver returns back corrupt data block(s) to the physical layer. Upon detecting a checksum violation, the physical layer responds to the read request with a checksum error.

The logical layer, on being notified of a checksum error, performs on-the-fly recovery using the redundant copy, and restores lost data onto the corrupt physical layer immediately. Table 2.1 shows the recovery time for various corruption cases. As can be seen, the recovery time is proportional to the amount of data lost within a file.

Graceful Degradation

Our test file system for graceful degradation consists of a collection of 12,000 32 KB files, organized uniformly across 100 directories. We created the test file system on a Loris installation with three disk drives. The number and size of files were chosen

to minimize the effect of caching, and the directory layout minimizes the effect of our linear file name lookup.

With this setup, we evaluated graceful degradation with two different file layout schemes, which we will detail shortly. However, in both cases, all directories are mirrored across the three disk drives, and all files are positioned in a fault-isolated manner. Directory replication is done by having the naming layer assign the RAID 1 policy to all directories. Fault-isolated file placement is done by storing each file, in its entirety, in at least a single physical module (as opposed to striping it across all modules). The ease with which we were able to provide these functionalities highlights the flexibility of a file-oriented stack.

Our workload is generated by a program that randomly picks a file and performs a 32 KB read, followed by a 32 KB write, overwriting the entire file. The program considers each open-read-write-close sequence as a single operation, and keeps track of the total number of successful operations.

Figure 2.5 illustrates graceful degradation under no replication. The files in this configuration are uniformly distributed across the three disk drives, that is, each corresponding physical module is responsible for serving a third of file requests. This is made possible by having the logical layer rotate file creation requests between the three physical modules. For instance, the create request for file 1 is forwarded to physical module P1, 2 to P2, 3 to P3, 4 again to P1, and so on.

As it can be seen, when the first fault occurs, the availability drops by roughly 33 %. This is expected since a third of the files are serviced by each physical module, and a device failure renders files serviced by that corresponding module inaccessible. A second failure results in another 33 % drop in availability. At this point, the directory hierarchy remains navigable (because directories are replicated on all drives), and the system continues serve requests that can be satisfied using the last disk drive. It can also be seen that the number of successful requests per second stays unaffected. The sharp drop in performance every thirty seconds is due to synchronous data and metadata flush during a sync.

Figure 2.6 shows graceful degradation, with files protected against a single disk failure. This case further illustrates the advantages of file-level RAID. A block-level RAID implementation would typically use RAID levels 1, 4 or 5 (two data and one parity) to protect against single disk failures. We used RAID 1 and we chose a layout that supported graceful degradation. It should be noted that the same technique can

Block type	**Affected**	**Recovery time**
Direct (actual data)	0.0039 %	28 ms
Single indirect	2.0000 %	157 ms
Double indirect	97.9727 %	6688 ms

Table 2.1: Recovery time after corruption of various data block types in a 100 MB file. For each block type the table lists: (1) the percentage of the file affected when a block of this type is corrupted, and (2) the recovery time measured after corrupting a block of this type.

Benchmark	MFS	Loris (single-process)			Loris (multiprocess)		
		No checksum	*Layout only*	*Checksum*	*No checksum*	*Layout only*	*Checksum*
PostMark	794.00 (1.00)	729.00 (0.92)	723.00 (0.91)	797.00 (1.00)	760.00 (0.96)	763.00 (0.96)	822.00 (1.04)
Applevel (copy)	79.86 (1.00)	86.41 (1.08)	86.00 (1.08)	101.36 (1.27)	94.66 (1.19)	94.38 (1.18)	109.63 (1.37)
Applevel (build)	58.06 (1.00)	58.50 (1.01)	58.48 (1.01)	60.21 (1.04)	73.68 (1.27)	73.78 (1.27)	75.61 (1.30)
Applevel (find and grep)	16.55 (1.00)	16.90 (1.02)	17.01 (1.03)	19.73 (1.19)	22.61 (1.37)	22.50 (1.36)	25.53 (1.54)
Applevel (delete)	19.76 (1.00)	10.01 (0.51)	10.01 (0.51)	13.06 (0.66)	18.68 (0.95)	18.96 (0.96)	22.00 (1.11)

Table 2.2: Transaction time in seconds for PostMark and wall clock time for applevel benchmarks. Table shows both absolute and relative performance numbers, contrasting MFS with Loris in several configurations.

Benchmark	Loris (without checksumming)				Loris (with checksumming)			
	RAID 0-4	*RAID 1-1*	*RAID 1-2*	*RAID 4-4*	*RAID 0-4*	*RAID 1-1*	*RAID 1-2*	*RAID 4-4*
PostMark	195.00	197.00	205.00	268.00	228.00	204.00	214.00	319.00

Table 2.3: Transaction time in seconds for different RAID levels. Each column RAID X-Y shows the performance of RAID level X in a Y-disk configuration.

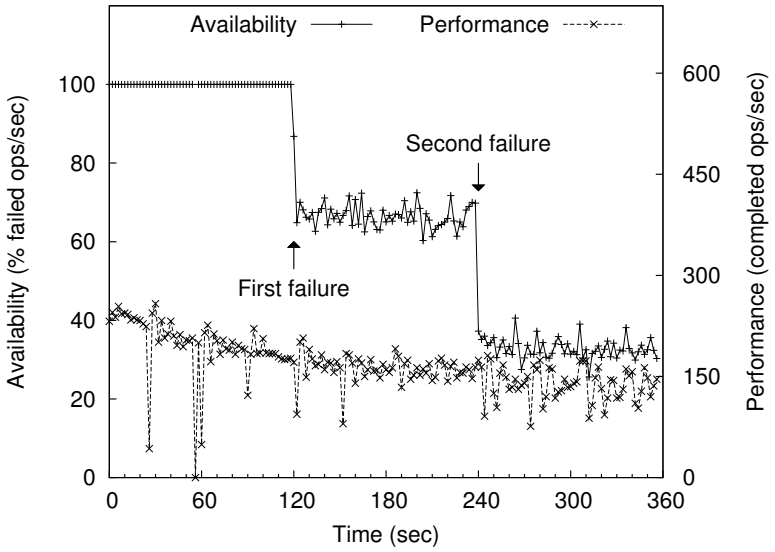


Figure 2.5: Graceful degradation case 1: Unreplicated files. The figure shows how Loris exhibits graceful degradation under unexpected failures. Each disk failure results in a third of files being inaccessible since files are not replicated. But the system continues to survive with an availability of around 33 % even after two disk failures.

also be used with other RAID levels in a file-level RAID.

With three disks, there are three possible combinations that can be chosen to protect a file against a single disk failure: D1-D2, D1-D3, and D2-D3. Our logical layer rotates files across these combinations. For instance, the first file is mirrored on disks D1-D2, second on D1-D3, third on D2-D3, fourth again on of D1-D2, and so on.

As it can be seen in Figure 2.6, the first failure has no effect on the system, as it would be with block-level RAID, since every file is protected against a single disk failure. A second failure would result in a block-level RAID implementation shutting down. In our case, we can see that the availability drops only by a third. All the files that were mirrored across the two failed disks are now inaccessible. Thus, even with just a single functional disk, Loris is able to maintain an availability close to 66 %.

We would like to point out that these techniques are complementary to higher levels of protection, that is, they can be used in combination with RAID 6 techniques, for instance, to build a file-level RAID array with extremely high reliability and availability. Furthermore, these techniques can be customized on a per-file basis, with different files using different levels of protection.

2.6.3 Performance Evaluation

In this section we present the performance evaluation of Loris. We first present an evaluation of the overhead of two important aspects of our new architecture: the parental checksumming scheme as implemented in the physical layer, and the process isolation provided by splitting up our stack into separate processes. We then present an evaluation of our file-level RAID implementation.

Checksumming and Process Isolation

We now evaluate parental checksumming and process isolation using two macrobenchmarks: (1) PostMark, configured to perform 20,000 transactions on 5,000 files, spread over 10 subdirectories, with file sizes ranging from 4 KB to 1 MB, and read/write granularities of 4 KB, and (2) an application-level macrobenchmark, which we will refer to henceforth as *applevel*, consists a set of very common file system operations including copying, compiling, running find and grep, and deleting.

We tested three checksumming schemes: no checksumming, checksumming layout only, and full checksumming. In the layout-only scheme, the CRC32 routine has been substituted by a function that always returns zero. This allowed us to measure only the I/O overhead imposed by parental checksumming. We ran these three schemes in both the single-process and the multiprocess Loris versions, yielding six

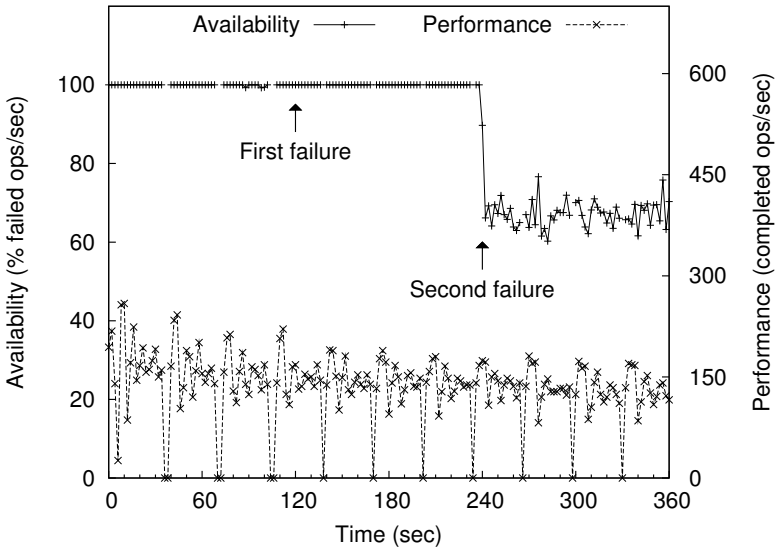


Figure 2.6: Graceful degradation case 2: Rotated mirroring of files. The figure shows how Loris exhibits graceful degradation with files protected against a single disk failure. Each of the three possible pairs holds a third of the files redundantly. As a result, the system continues to serve two-thirds of all its files, with an availability of 66 %, even under two disk failures.

configurations in total.

Table 2.2 shows the performance of all six new configurations, compared to the MFS baseline. Loris outperforms MFS in most configurations with PostMark. This is primarily due to the fact the delete algorithm used by Loris performs better than MFS, as seen in the applevel delete benchmark in Table 2.2.

Looking at the applevel results, it can be seen that the single-process case suffers from an overhead of about 8 % compared to MFS during the copying phase. This overhead is due to the difference in caching logic between Loris and MFS.

The block-level buffer cache in MFS makes it possible to merge and write out many physically contiguous files during sync periods. Since Loris has a file-level cache, it is unaware of the physical file layout and hence might make less-than-optimal file evictions. In addition, our prototype also limits the number of files that can be written out during a vectored write call, to simplify implementation. These two factors result in a slight overhead, which is particularly noticeable for workloads with a very large number of small files.

Since the copy phase involves copying over 75,000 files, of which a significant percentage is small, there is an 8 % overhead. Even though the overhead is small, we plan on introducing the notion of *file group identifiers*, to enable the passing file relationship hints between layers. The cache layer could then use this information to evict physically contigous files during a vectored write operation. This and several other future optimizations should remove this overhead completely.

Another important observation is the fact that in both single-process and multiprocess configurations, the checksum layout incurs virtually no overhead. This means that the entire parental checksumming infastructure is essentially free. The actual checksum computation, however, is not, as illustrated by a 7 % overhead (over no checksum case) for PostMark, and 3-19 % overhead in applevel tests.

It should be noted that with checksumming enabled, *every* file undergoes checksum verification. We would like to point out that with per-file policy selection in place, we could reduce the overhead easily, by either checksumming only important file data, or by adopting other lightweight verification approaches as opposed to CRC32 (such as XOR-based parity). For example, we could omit checksumming compiler temporaries and other easily regeneratable files.

We also see that the multiprocess configuration of Loris suffers consistently, with an overhead ranging between 11-45 %. This contradicts the results from PostMark, where the multiprocess case has an overhead of only about 3 % compared to the single-process case. This is again due to the fact that the applevel benchmark has a large number of small files compared to PostMark. Data copying and context switching overheads constitute a considerable portion of the elapsed time when small files dominate the workload. With large files, these overheads are amortized over the data transfer time. We confirmed this with separate microbenchmarks, not shown here, involving copying over a large number of small files.

File-Level RAID

In this section, we evaluate our RAID implementation. We test two RAID 1 configurations: (1) RAID 1 on a single disk, and (2) RAID 1 with mirroring on 2 disks. The RAID 0 and RAID 4 implementations use all four disks, with RAID 0 configured to use 60 KB stripe units, and RAID 4 80 KB stripe units for all files. These stripe sizes align full stripe writes with the maximum number of blocks in a vectored write request (240 KB).

We ran PostMark in a different configuration compared to the earlier benchmark—20,000 transactions on 60,000 files, distributed across 600 directories, with file sizes ranging from 4 KB to 10 KB. Small-file workloads are challenging for any RAID implementation since they create lots of partial writes. We chose this benchmark to evaluate how our file-level RAID implementation handles small files.

The most important observation from the PostMark results is that RAID 4 is much slower than RAID 1 with mirroring. The three main reasons for this slowdown are as follows. (1) RAID 4 suffers from the partial-write problem we mentioned earlier. It is important to note that such partial writes would translate into partial stripe requests in a block-level RAID implementation. (2) Parity computation in RAID 4 is expensive compared to mirroring in RAID 1. Both block and file RAID implementations share these two problems. (3) Our implementation of RAID 4 negates the advantages of vectored writes for small files.

To illustrate this problem, consider a vectored write request at the logical layer. The algorithms used to process this request in our implementation are very similar to a block-level RAID one. Each request is broken down into individual file requests, which are further divided into constituent stripe requests, and each stripe request is processed separately. In our current prototype, we implemented sequential processing of stripe requests. Thus, if the workload consists of many small files, a vectored write gets translated into single writes for each file, negating the benefit of vectoring the write request.

The solution to all the aforementioned problems is simple in our case. Parity amortizes the cost of redundancy only when write requests span multiple stripe units. Thus, we are better off using RAID 1 for small files. As our RAID implementation is file-aware, we can monitor and collect file read/write statistics, and use it to (re)assign appropriate RAID levels to files. Matching file access patterns to storage configurations is future work.

2.7 Conclusion

Despite dramatic changes in the storage landscape, the interfaces between the layers and the division of labor among layers in the traditional stack have remained the same. We evaluated the traditional stack along several different dimensions, and highlighted several major problems that plague the compatibility-driven integration

of RAID algorithms. We proposed Loris, a file-level storage stack, and evaluated both reliability and performance aspects of our prototype.

3

Integrated System and Process Crash Recovery in the Loris Storage Stack

Abstract

In this paper, we look at two important failure classes in the storage stack: system crashes, where the whole system shuts down unexpectedly, and process crashes, where a part of the storage stack software fails due to an implementation bug. We investigate these two problems in the context of the Loris storage stack. We show how restoring metadata consistency can provide a common first step for recovery from both types of crashes. In addition, we present fine-grained and corruption-resistant data resynchronization as the second step for system crash recovery, and an in-memory roll-forward log that can provide strong guarantees as the second step for process crash recovery in a microkernel setting. We implement our findings in our Loris prototype, and implement a new crash-resistant on-device layout as part of our proof of concept. The evaluation shows that our approach provides increased reliability at a reasonable performance cost.

3.1 Introduction

In virtually any computer system, there is a component responsible for storing users' data: the storage stack. A large part of the storage stack is software (usually) in the operating system. While it is important that the storage stack has proper performance, we argue that reliability is at least as important. After all, not dealing with storage stack failures can translate directly into data loss. In this paper, we look at two threats: system crashes, and storage stack software (process) crashes.

A system crash is a whole-system failure of a machine. The causes of such failures include power outages, hardware failures, and operating system kernel crashes. In the event of a system crash, the storage stack does not get the opportunity to write out dirty data in memory, or even complete the current operation. This may result in inconsistent on-device data structures, which could lead to failure to reload these data structures from disk after the system has restarted. That in turn could lead to data loss.

Another major reliability threat comes from software bugs. Previous research has suggested that the number of bugs in software is roughly linear in its number of lines of code [111]. A full-fledged operating system storage stack can easily consist of hundreds of thousands of lines of code, and this makes the presence of many bugs highly probable. Any such bug has the potential to subvert the proper operation of the storage stack and, again, cause data loss.

In previous work, we have designed a new storage stack called Loris [9]. This storage stack has advantages in the areas of reliability, heterogeneity, and flexibility. In this paper, we investigate how to add support in Loris for system crash recovery, and for process crash recovery from transient failures in the lower layers of the stack, without compromising Loris' other reliability guarantees.

We show that the recovery procedure for both types of crashes require a single shared first step, namely restoring consistency of all metadata maintained internally by the storage stack. We argue that the storage stack should incorporate first-class support for this, and to this end we add the concept of global consistent *checkpoints* to Loris.

After this shared first step, each of the crash recovery procedures requires a different second step. For system crashes, recovery involves restoring proper redundancy of (user) data, using a resynchronization procedure similar to that of traditional software RAID. We present a data resynchronization approach that is both fine-grained and corruption-resistant. For process crashes, recovery involves an in-memory log to roll forward from the last checkpoint to the current state. We employ this approach in a microkernel environment to provide better recovery guarantees than any previous work.

We implement these ideas in our Loris prototype. As part of this, we present a new crash-resistant device layout and a corresponding software implementation called "TwinFS." We evaluate our work using performance benchmarks and reliability tests, aiming to prove that our design can be adopted in environments where a

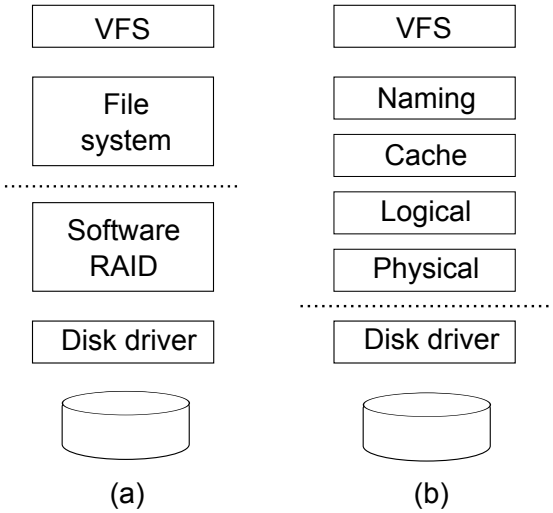


Figure 3.1: The figure shows (a) the layers of the traditional stack, and (b) the new arrangement in Loris. The layers above the dotted line, and only those, are file-aware.

moderate performance overhead is acceptable, but high reliability is a requirement.

The rest of the paper is organized as follows. Sec. 3.2 describes the Loris storage stack that we developed previously. In Sec. 3.3, we describe the two main problems to address, and we sketch an architecture that integrates a solution for both. In Sec. 3.4–3.6, we present the design and implementation of the three parts that make up the solution. In Sec. 3.7, we evaluate the prototype. Sec. 3.8 covers related work. Sec. 3.9 concludes and lists future work.

3.2 Background: the Loris storage stack

The traditional storage stack as found in most operating systems is shown in Fig. 3.1a. A Virtual File System (VFS) layer multiplexes application calls across file systems. File systems are generally designed to operate on one device, although a software RAID layer may transparently add storage redundancy using multiple devices below it. The actual devices are controlled by disk driver software.

The Loris stack was formed by first splitting the traditional file system into three layers (naming, cache, and layout), and then swapping the layout layer (also called the *physical* layer) and the traditional software RAID layer (forming the *logical* layer). The VFS and disk driver layers are left unchanged. The result is depicted in Fig. 3.1b.

The Loris stack is completely file-oriented: the four layers communicate in terms of *files* only. Each file has a unique file identifier, and a small set of attributes associated with it. The layers use and implement the following operations: create, delete, read, write, truncate, getattr, setattr, and sync.

Compared to the traditional stack, the Loris stack has reliability, heterogeneity, and flexibility advantages [9]. We have built a Loris prototype on the MINIX 3 microkernel system [62], where all layers and file stores are separate user space processes. We will now describe the four layers.

3.2.1 Layers of the stack

At the bottom, the **physical layer** consists of one or more **file stores**. Each file store manages one underlying device, and maintains the layout on that device. It exposes an independent set of *physical files*, each with a physical file ID chosen by the file store. Each file store has a small local cache for the metadata specific to that file store, which we call "layout metadata." All file stores are required to implement *parental checksumming* in their layout [9]. As a result, they can reliably detect all whole-device failures as well as any form of (overt and silent) corruption.

Our prototype implements one file store called "PhysFS," based on the traditional UNIX file system. In PhysFS, the layout metadata structures form a virtual tree. All parents in this hierarchy point to their children by means of *safe block pointers*. A safe block pointer consists of the block number of the child block, and a checksum of its contents. File metadata, including attributes, are stored in *inodes*. Inodes use safe block pointers to point directly to data blocks, and to indirect blocks that contain (safe) pointers to either data blocks or other indirect blocks. Free inodes and blocks are tracked using inode and block *bitmaps*. The parental checksumming hierarchy is completed with three special inodes that point to the blocks of the inode area and bitmap areas. These inodes are stored in a *root block*, which is self-checksummed and forms the root of the metadata tree.

The inode, bitmap, and root block metadata areas are preallocated and statically sized. Out of all the layout metadata, only indirect blocks are allocated dynamically, and stored together with data blocks in the data area.

The **logical layer** implements a file-based version of RAID, providing the abstraction of *logical files*. Each logical file is made up of one or more physical files on different file stores. The logical layer multiplexes operations across the file stores in a RAID-like fashion. For example, a two-way mirrored logical file is stored as two identical physical files on different file stores (and thus devices). The logical layer keeps a *mapping*, which for each logical file specifies: the RAID level, the corresponding file stores and physical file IDs, and other RAID parameters such as the stripe size. The logical layer stores the mapping in a special file that is mirrored across all file stores. This file is part of the global metadata of the Loris stack, which we call "stack metadata."

The logical layer also implements RAID-like recovery mechanisms. If any of the file stores report a checksum error, recovery is attempted. In case of permanent failure, operations will continue to be served as long as enough redundancy is available. Redundancy guarantees follow the standard RAID failure model [113], although on a per-file basis.

The **cache layer** implements in-memory caching of logical file data. It uses a large amount of system memory for caching the contents of files.

The **naming layer** provides a POSIX abstraction by translating VFS operations to Loris operations. This layer implements directories, which are stored using Loris files. Lower layers are only aware that these directory files are part of the stack metadata. The naming layer uses Loris' file attributes to store POSIX attributes. It is responsible for picking logical file IDs for new files, and for tracking open deleted files.

3.3 The case for integrated recovery

In this section, we present the two challenges that we would like to address in the Loris storage stack: recovery from system crashes (Sec. 3.3.1) and process crashes (Sec. 3.3.2). We then show that we can exploit significant overlap between the solutions to both problems (Sec. 3.3.3).

3.3.1 Recovering from system crashes

Metadata consistency

In the traditional storage stack, the file system typically implements a consistency scheme. Such a scheme returns the on-device structures to a consistent state after a system crash. Some limit themselves to metadata for performance reasons; others also cover the user data. Well-known schemes include journaling [53], logging [112], copy-on-write (CoW) [66], and soft updates [43].

All such schemes can be roughly described as periodically establishing consistent restore points, *checkpoints*, that can be reloaded such that any potentially inconsistent changes made after it are discarded upon system crash recovery. For example: copy-on-write schemes do this by writing a new root block; logging and journaling schemes do this by writing a commit record, and soft-update schemes effectively create a new checkpoint upon every metadata write.

In the Loris stack, each file store is free to implement a layout tailored to its underlying device. Layout metadata structures (such as inodes) are thus managed on a per-device basis. Consistency of these structures must therefore be managed on a per-device basis as well. Thus, each file store is necessarily responsible for managing its own *local checkpoints*.

However, it is not enough for each file store to restore its local layout metadata to just any consistent state after a system crash. The higher layers of the stack rely on the file stores in the physical layer to properly store stack metadata. For example, it is crucial for the stack that all mirrored copies of directories and the mapping are in sync and their contents are consistent with the file stores. Thus, the first step towards system crash recovery is restoring *global metadata consistency* across all the file stores, which includes both layout metadata and stack metadata.

There are two different approaches that we can adopt for this step. The first is to allow file stores to take local checkpoints whenever they choose, and then bring them back in sync at restore time. However, this approach imposes several requirements. For example, bringing the file stores back in sync can only be done if the file stores keep on-device logs that allow them to roll each other back or forward as appropriate. This rules out consistency schemes that do not keep such a log (e.g., copy-on-write). In addition, the file stores would have to become aware of consistency requirements for stack metadata updates, for example between file creates and directory writes, so as to create consistent local checkpoints. This effectively imposes stack-wide support for atomic transactions.

A better alternative is to globally coordinate the creation and reloading of checkpoints throughout the whole stack, and across all file stores. That means that all file stores were in sync at the time that the local checkpoints were taken, and they are thus again in sync if the same checkpoints are reloaded after a system restart. In Loris, we can extend the **sync** call to establish such *global checkpoints*. The whole stack is involved, so all layers get the chance to flush any pending stack metadata changes. The only downside is that the sync call must be a stack-wide barrier operation: while the checkpoint is being taken, any new state changes could subvert its consistency. Overall, this approach is preferable because it is simple to implement, and gives each file store a large freedom in choosing a local consistency scheme that is optimal for the underlying device.

Data resynchronization

The checkpointing system protects metadata, but for performance reasons it may not fully cover user data. A system crash may thus cause inconsistency between redundantly stored copies of the same data, due to a partially completed multidevice write operation. For example, a data write call to a mirrored file may make it to one mirror before a crash, but not to the other.

Traditional software RAID faces the same issue. For this reason, it typically implements *resynchronization*, whereby all blocks from the devices are read in and checked to find and fix any cross-device inconsistencies after a system crash. A similar resynchronization process is also necessary for user data in Loris. Thus, the second step towards system crash recovery is resynchronizing data.

However, traditional software RAID resynchronization suffers from two major problems. First, the RAID layer has no knowledge about the file system or even about liveness of blocks, and thus has to resort to scanning all devices in their entirety, including all metadata, data, and unused blocks. This may take a prohibitively long time–in the order of magnitude of hours to days. Second, the RAID layer cannot tell whether an inconsistency is the result of the system crash or data corruption. It may thus restore RAID consistency by overwriting a valid copy of data with a corrupted one, causing data loss. This is known as the RAID "write hole." As we show later, we can significantly improve the approaches to solving both problems in

the Loris storage stack.

3.3.2 Recovering from process crashes

In a microkernel environment, most operating system components are implemented as user space processes. Each such system process has its own address space and restrictions on inter-process communication (IPC), and as such, is an individual failure domain. We use the term *process crash* to describe an observable failure in a system process. A process is said to "crash" when it performs an illegal CPU or memory operation, does not respond in time to periodic ping signals, performs disallowed IPC, or exits prematurely, for example due to a failing `assert`. If the cause of the crash is transient, repeating the operations leading up to the crash may not result in a crash the next time. We only consider transient crashes.

MINIX 3 provides detection of process crashes and supports basic recovery. When a process crashes, a fresh instance of the process is started. The internal state of the crashed process is lost. This approach works well for MINIX 3's device driver processes (including the disk drivers in the storage stack), which have little to no internal state [62]. However, other system components need to have their internal state fully restored before they can continue normal operation.

In this work, we focus on process crash recovery of the lower two layers of the Loris stack: the logical and physical layers. These layers form the largest part of the entire stack, and typically contain large amounts of in-memory state–mainly metadata structures that have been updated in memory as a result of application calls, but have not yet made it to disk. We discuss the other layers in Sec. 3.9.

Our goal here is twofold. First, we want to provide recovery that is fully transparent to applications. This means that system calls must not be aborted with an error due to a process crash, and that the effects of earlier system calls must never be lost. Second, we want to make no assumptions about what has happened *inside* the crashed process prior to the crash. Thus, recovering the state from the crashed process memory image is not an option, as this state may have been corrupted.

It is too costly to make a second in-memory copy of all process state and changes to it; it is even more costly to constantly flush the latest state to disk. However, it is possible to perform recovery using on-disk and in-memory state in combination. A large part of the required state is on the device at any time, and the necessary remaining part can be kept in memory until it is stored on disk. The recovery procedure then consists of first reloading a previous state from disk, and afterwards replaying from memory any state changes that have been made since.

3.3.3 Integrated recovery

We now show how both the recovery approaches can share the same first step. The checkpointing system that forms the basis for system crash recovery can be used to provide the first step toward process crash recovery as well. Thus, after a process

crash, the recovery starts by rolling back the logical and physical layers to the latest checkpoint, discarding any state modified since then. The second step then consists of rolling forward these layers from the latest checkpoint to the current state.

In the next three sections, we present the design and implementation of this overall approach in Loris. We explain each of the three necessary parts: restoring global metadata consistency with checkpointing (Sec. 3.4), data resynchronization for system crash recovery (Sec. 3.5), and in-memory roll-forward logging for process crash recovery (Sec. 3.6).

3.4 Checkpointing

We will now describe the design and implementation of the checkpointing system in Loris. As outlined, this system establishes *global* checkpoints by coordinating the file stores' creation of *local* checkpoints. For this work, we have developed a new file store called "TwinFS," which implements a new on-device layout with support for local checkpoints. We start by describing this file store (Sec. 3.4.1). Then, we define the requirements for file store consistency schemes in general (Sec. 3.4.2). Finally, we describe the procedures for establishing and reloading global checkpoints (Sec. 3.4.3).

3.4.1 The TwinFS file store

TwinFS is based directly on our original PhysFS file store implementation as described in Sec. 3.2.1. It adds the concept of checkpoints, by employing a copy-on-write-like scheme for the blocks that are part of those checkpoints. We call such

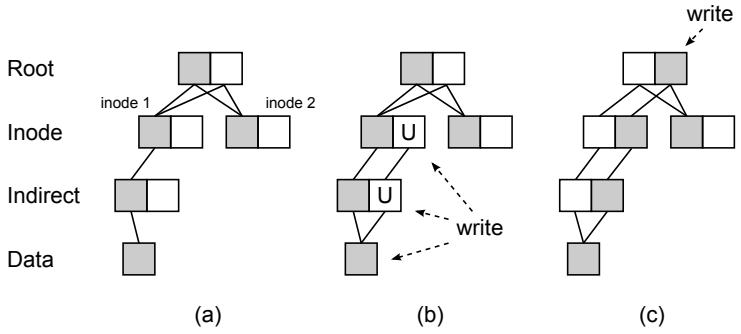


Figure 3.2: TwinFS hierarchy example. Each block's left and right twins are shown; each latest stable twin is grayed. The lines represent safe block pointers. As simplification, each inode block contains only one inode. In (a), inode 1 has one indirect block pointing to a data block. In (b), this data block is overwritten in-place, and its ancestors are updated with new checksums by writing to the unstable (U) twins–up to but excluding the root block. In (c), a new checkpoint is established by writing a new root block.

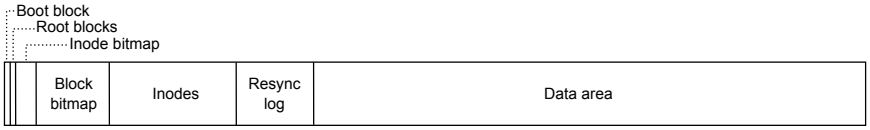


Figure 3.3: High-level overview of the TwinFS on-device layout. The root, inode, and bitmap blocks are metadata and thus protected. The data area contains both protected and unprotected data blocks, as well as indirect blocks, which are metadata and thus protected. The resynchronization log is described in Sec. 3.5.3.

blocks *protected*. At the very least, protected blocks are used to store all the stack and layout metadata.

Each protected block has two preallocated on-device locations ("twins"). At any time, one of these locations is used to store the "stable" version of the block: the block as it was at the time of the last checkpoint. The other is used to store the "unstable" version of the block: the most current version, which may be updated several times before the next checkpoint is taken. When a checkpoint is taken, the roles of all modified blocks are swapped: the unstable twin is marked stable, and the previously stable twin will be used to store subsequent block updates. Unmodified blocks are left untouched, so not all blocks switch twin sides between each subsequent checkpoints. An example is shown in Fig. 3.2.

In theory, each of the two twins could be located anywhere on the device. We simplify our implementation by hardcoding the block distance between each of the two twins to a static *twin offset*. This way, we can represent the *twin state* of each block using just two bits: one bit that identifies the last-modified twin (left or right), and one bit that identifies whether the block has been modified since the last checkpoint. The latter keeps the file store from having to track in memory which blocks have been modified since the last checkpoint (and thus have already switched sides).

The two-bit twin state of each block is stored in the *safe block pointer* in its parent. Marking a block as unstable thus implies recursively marking all parent blocks as unstable, all the way up to the root of the hierarchy. Exactly the same already happens in PhysFS due to updating the parental checksums. Hence, updating the blocks' twin state introduces no extra overhead. Note that unlike with true copy-on-write schemes, the preallocation of the twins guarantees that no new blocks need to be allocated in this process.

The overall TwinFS on-device layout is shown in Fig. 3.3. All layout metadata blocks are protected using the "twinning" scheme. In the statically allocated metadata areas, this results in a repeated pattern of N left twins being followed by N right twins, were N is the twin offset. These metadata areas are doubled in size as a result. Indirect blocks in the data area are twinned as well. TwinFS can also protect the data blocks of selected files. First and foremost, this data block protection is applied to the stack metadata files (logical mapping; directory files), since these must be included in the checkpoints as per Sec. 3.4.2. As stated before, file data of

important user files can also be protected with this approach; however, it would be costly (in performance and space usage) to protect *all* file data this way. Thus, in the data area, the twin pairs of indirect blocks and protected data blocks are interspersed with unprotected data blocks. When a protected block is created in the data area, two physical disk block locations at a fixed separation must be allocated.

The root block of the layout scheme is "twinned" as well. Writing out a new root block equals taking a new checkpoint, and this is done only as part of a **sync** call. The root block is self-checksummed and contains a timestamp. A write cache flush is performed on the underlying device both before and after writing out the root block. Since the twin state of the root block cannot be stored anywhere, TwinFS has to read in and verify the checksum of both root block twins at startup.

3.4.2 General consistency scheme requirements

During recovery, all file stores must agree on the checkpoint to reload. For some file stores, this may be the penultimate checkpoint they have taken: it is possible that a system or process failure occurs during the checkpointing operation, whereby some file stores have finished establishing the checkpoint, and others have not. This results in the following requirements for the consistency schemes employed by the file stores: 1) it must always be possible to recover all (layout and stack) metadata to the state in the last checkpoint; and, 2) right after taking one checkpoint locally, but before this checkpoint has been finished across all file stores, it must remain possible to restore the *previous* checkpoint as well.

TwinFS meets these requirements. At any time, one of the root block twins identifies a stable checkpoint, since all of the metadata locations it refers to are stable and thus are left intact at least until after taking the next checkpoint. Immediately after taking a new checkpoint, both root block twins identify usable checkpoints, and either can be reloaded. Once new "unstable" data has been written out, only the latest checkpoint can be reloaded.

Even though we use only TwinFS in our prototype, each file store has the freedom to employ *any* consistency scheme that satisfies the stated requirements. This includes several of the more well-known consistency schemes:

- Copy-on-write consistency schemes effectively implement checkpoints by persisting new root nodes of the metadata tree. A minimum of two root nodes is enough.

- For journaling schemes, each *transaction* has to span between two checkpoints, and taking a checkpoint amounts to committing the current transaction. Copying from the journal to the original location may commence only once taking the checkpoint has finished globally.

- Logging schemes could be used as is, as long as the recovery procedure restores the latest checkpoint without performing any roll-forward on metadata.

- However, soft update schemes rely on frequently updating metadata in-place. This makes rollback impossible. Any such scheme is not suitable.

For consistency schemes that can overwrite data blocks in-place, such as some forms of journaling, there is an additional requirement: a data block must never be overwritten with contents of one file, if according to the last checkpoint this block was assigned to another file. If this were allowed, the new data could be read from the old file after a checkpoint restore, which could constitute a security violation. While parental checksumming helps to protect against this case, it does not provide a secure solution. In TwinFS, blocks that are freed are not reused until a new checkpoint is taken. This equally applies to protected and unprotected blocks.

3.4.3 Taking and reloading checkpoints

New global checkpoints are established using the Loris **sync** operation, which travels from the naming layer down the stack. This call causes all layers to flush pending data and stack metadata updates, and tells the file stores to create a new checkpoint. Sync calls are initiated upon application request (with POSIX' *sync* or *fsync*), periodically, and at system shutdown. During the sync operation, other state-changing Loris operations are deferred.

Upon startup, the logical layer queries all file stores for valid checkpoints. In response to this, all file stores return the timestamps of their valid checkpoints. If the system shut down cleanly, or crashed while no sync call was ongoing, all file stores will share the same latest checkpoint timestamp. If the system crashed during a sync call, not all file stores may have the latest checkpoint, but they will all have the penultimate one. After all file stores have reported their available checkpoint timestamps, the logical layer instructs them to load the most recent common checkpoint.

3.5 Data resynchronization

Some consistency schemes include *all* data in the checkpoints. Examples would be a pure copy-on-write or log-structured layout. Journaling and twinning layouts may or may not. For those that do not, data resynchronization may be needed to restore full consistency after a system crash.

In Sec. 3.3.1, we listed two problems in traditional RAID resynchronization. We now show that the Loris stack offers the opportunity to improve on both problems: the large area to scan (Sec. 3.5.1) and the possibility of corruption (Sec. 3.5.2). We then describe how TwinFS implements support for data resynchronization (Sec. 3.5.3), and we outline the full resynchronization procedure (Sec. 3.5.4).

3.5.1 Limiting the areas to scan

In the Loris stack, we can very narrowly define the areas to which data resynchronization should be applied. First of all, since the file stores are completely file-based, unused parts of the disk are inherently excluded from resynchronization. Furthermore, resynchronization only involves data files. The layout metadata and the stack metadata files are already covered by the checkpointing system.

Second, Loris' per-file policies allow certain files to be stored with more redundancy than others. Files that are stored on one device only, need not be resynchronized. Moreover, the per-file policy system allows the user to assign a level of *importance* to a file. Files that are deemed especially important by the user, can be included in the checkpoints. This excludes them from resynchronization.

Finally, resynchronization is needed only for data blocks that have been overwritten in-place since the last checkpoint. After all, not-in-place updates are simply discarded when restoring a checkpoint. Since the security requirement from Sec. 3.4.2 imposes that no file data can be overwritten with another file's data, resynchronization can be limited to in-place overwrites within the *same* file.

3.5.2 Verifying data

If a data block is overwritten in-place, and the last checkpoint is restored afterwards, the parental checksum of the block will no longer match. The file store can then no longer discern whether the block was merely overwritten, or has been subject of corruption. Thus, it has to either discard the data block, resulting in data loss, or ignore the checksum, risking to pass corrupted data to the application. For a reliable stack, neither is acceptable.

The file store must therefore make sure that when a data block is overwritten in-place, its new parental checksum has already been written to disk. The file store thus has to persist this information outside the checkpoints. This can be done by means of a log of "checksum records." Depending on the consistency scheme, this may be a dedicated resynchronization log, or be integrated in the main journal or main log. Before a data block is overwritten, the file store persists a record that contains the new checksum for the block. After restoring the last checkpoint, the file store can scan the log for these records.

The checksum records have two purposes. First, it allows the file store to tell whether the contents of an overwritten data block are valid or corrupted. The block contents are considered valid if the block checksum matches one of the recorded checksums. After all, a system failure may occur between writing the record and writing the data block, in which case an earlier checksum is still valid. This checksum is possibly in an earlier checksum record and otherwise in the last checkpointed copy of the inode.

Second, it allows the file store to tell exactly which blocks have been overwritten and thus should be subject to resynchronization. This limits resynchronization ex-

actly to the areas defined in the previous section. However, even though files that are stored without redundancy also require no resynchronization, checksum records also have to be generated for those files, because the corruption concern applies equally to them.

If a system crash resulted in a torn write of a data block, none of the checksums will match. This case cannot be distinguished from other forms of corruption, and the data block will be lost. Given sufficient redundancy, it can be restored from other file stores. In the worst case however, every device suffers from a torn write. By including their file data in the checkpoints, stack metadata files and important user data files are protected from this problem.

3.5.3 The TwinFS resynchronization log

Since TwinFS does not include all data blocks in its checkpoints, it may end up overwriting unprotected data blocks in-place. Thus, TwinFS must perform data resynchronization. To this end, we add a dedicated resynchronization log to it, which uses a reserved device area to store checksum records.

Each checksum record contains a physical file ID, a file block offset, and the new block checksum. Pending checksum records are aggregated into "log blocks," which are self-checksummed and contain the corresponding checkpoint timestamp. In order to let file stores generate and aggregate checksum records ahead of the actual write operations, we add a **prewrite** Loris call. This call is sent down from the cache to the physical layer when an application performs a write call, providing an early copy of the data to the file stores. Whenever TwinFS receives a write operation that overwrites unprotected blocks, it ensures that the corresponding log blocks have been flushed to the device first.

After a checkpoint has been reloaded, TwinFS goes through the log area, and processes log blocks that have both a valid checksum and a matching checkpoint timestamp. It performs two actions on the records in each valid log block. First, it compares the checksum in the record to the computed checksum of the corresponding data block. If those match, it updates the data block's safe block pointer (in the file inode or an indirect block) with this checksum, effectively marking the data block as not corrupted. Second, it reports the file byte range from the record to the logical layer, for the purpose of resynchronization across file stores.

3.5.4 Resynchronization procedure

After the file stores have checked local data checksums, the next step is resynchronization *across* file stores. After all, the redundantly stored copies may still be out of sync.

The resynchronization procedure is performed by the logical layer as part of system crash recovery, right after restoring the appropriate checkpoint. The logical layer queries all file stores for a list of physical file IDs and byte ranges that are

to be resynchronized. In the worst case, resynchronization involves all data from all unprotected, redundantly stored files; in the common case, only a few files and blocks will be involved. File stores that include all data in their checkpoints always report an empty set.

The logical layer then maps each reported (physical) file ID to a logical file ID. We do this by storing the logical file ID as an attribute of each physical file. The logical layer reads in the reported byte ranges from all file stores involved in storing this logical file, and optionally writes back resynchronized data to some of them. In the case of (RAID1-like) mirroring, all mirrors are synchronized to the contents from the first file store that does not report a checksum error. In the case of (RAID4/5-like) parity-striping, the parity is recomputed, unless one of the file stores reports a checksum error. In that case, that file store's contents are recomputed. If more checksum errors are reported than supported by the RAID failure model, the affected byte ranges are marked as bad, and will result in an error being returned to the application when being read later.

By disregarding data copies with checksum errors before performing resynchronization across file stores, we solve the write hole problem. We do not offer guarantees about which valid data copy is restored, however. Content-level data consistency is thus left to applications, as with standard RAID.

3.6 In-memory roll-forward logging

The process crash recovery procedure consists of two steps: first restoring the last checkpoint, and then rolling forward the lower layers by replaying operations. The cache layer, which we currently assume to be a stable point in our stack, performs the second step, by keeping an in-memory log of operations. We describe how this log interacts with the checkpointing system (Sec. 3.6.1), the operation of logging and replay (Sec. 3.6.2), and the resulting assumptions and guarantees for process crash recovery (Sec. 3.6.3).

3.6.1 Interaction with checkpointing

When any of the processes in the lower two layers crashes, we choose to restart *all* of them, including all file stores. This has two advantages. First, this saves us from adding extra custom recovery code in those processes. Upon their restart, they will simply cooperate in restoring the latest checkpoint as part of their normal startup procedure. Second, there are no corner cases to handle when multiple processes crash at once.

The logical layer can detect when any file store or itself has restarted. First, when a Loris process is started, it will report its presence to the next layer up the stack. An unexpected presence notification from a file store thus indicates that a file store has restarted. Second, when a MINIX 3 system process crashes, it is restarted with a flag indicating that it crashed. If the logical layer itself restarts, this flag will be set.

Upon detecting a file store crash, the logical layer commits suicide (gracefully), forcing itself to restart. When it comes back up after a restart, the restart flag will be set; this may be the a result of either a local crash or committing suicide. The logical layer then kills and thus restarts all file stores. The result is that regardless of where a crash (or multiple concurrent crashes) happened, both the logical and physical layers will end up being restarted with a fresh state.

After that, the logical layer will perform the standard checkpoint reloading procedure as part of its startup procedure. However, when the logical layer's restart flag is set, it skips the unnecessary data resynchronization phase: any in-file overwrites that took place after taking the latest checkpoint but before the crash, will be performed again.

3.6.2 Logging and replay

At all times, the cache keeps an in-memory log of operations performed after taking the last checkpoint. This log is cleared upon each successful sync call. All Loris operations that modify state in the lower layers are stored in the log: create, delete, write, truncate, and setattr. Since the cache uses pages as the smallest unit of storage, the new version of an entire page is stored as part of the log entry for a Loris write operation. Multiple writes to the same page are merged, and pages are removed from the log as appropriate upon Loris truncate and delete calls. As long as a page is in the main cache, the log only keeps a pointer to it; a copy is made for the log when the page is evicted from the main cache. If the total size of the log exceeds a configurable threshold, the cache makes an upcall to the naming layer to trigger an early sync. Note that the naming layer's subsequent flush will first cause the log to expand further (but see Sec. 3.9 on future work).

After startup, the logical layer always announces its presence to the cache layer. The cache layer can tell from an unexpected presence announcement that the logical layer has restarted. The cache layer then cancels all ongoing downcalls, replays the in-memory log by issuing all operations in the log in sequence, and, upon success, restarts the previously ongoing calls. None of this is exposed to applications in any way. If the replay procedure fails, it is retried for a predefined number of times. Upon consistent failure, application-transparent recovery becomes impossible, and the entire stack is shut down to prevent further data loss.

There is one exception. It may happen that the system crashes before a sync call completes, but after all file stores have established a new checkpoint. The newer checkpoint would then be reloaded, causing unexpected failures during replay. For this reason, the logical layer informs the cache layer about checkpoint timestamps, and upon a mismatch after a crash, the cache clears its log instead of replaying it.

Benchmark	PhysFS	PhysFS+df	TwinFS-0	TwinFS-4	TwinFS-8	TwinFS-16	TwinFS-32
PostMark (transaction time, sec)	1097	1086	1101	1192	1158	1144	1150
FileBench File Server (ops/sec)	349.53	350.25	372.18	343.78	344.16	344.16	343.47
FileBench Web Server (Zipf) (ops/sec)	549.06	548.01	571.78	535.94	536.04	540.82	543.57
OpenSSH build (sec)	618.78	620.20	629.23	630.96	630.48	630.41	631.88

Table 3.1: Transaction time in seconds for PostMark (lower is better), operations per second for File Server and Web Server (higher is better), and wall clock time for OpenSSH build (lower is better). Performance is shown for PhysFS, PhysFS with delayed freeing, and TwinFS with various twin block offsets.

3.6.3 Assumptions and guarantees

Due to the process isolation offered by the microkernel environment, the only way in which failures can propagate, is through inter-process communication between processes. Moreover, since we completely restart the logical and physical layers, we throw out all of their internal state, including any state that has been corrupted as part of the failure.

As a result, we make only two assumptions about the behavior of any failing process: 1) no "bad" (corrupted) results are passed up to the cache; 2) no bad (meta)data may be written to the devices, unless they are discarded again once a checkpoint is restored. The second point implies that writing out corrupted unstable blocks is allowed in the failure model, as long as a crash happens before these blocks are made stable. This facilitates performing internal integrity checks before taking a new checkpoint.

As long as the two assumptions are not violated, this approach guarantees proper recovery from *any* bad behavior in the lower layers. This includes: arbitrary memory overwrites (wild writes), including heap and stack corruption; arbitrary function calls; infinite loops; and, arbitrary allocation of resources available to the processes, including memory.

3.7 Evaluation

We now present a performance and reliability evaluation of our prototype. All of the following experiments were conducted on an Intel Core2Duo E8600 PC, with 4GB of RAM, and two 500GB 7200RPM Western Digital Caviar Blue (WD5000AKS) SATA hard drives for testing purposes. The tests were run on the first 8GB of the disks. All benchmarks were run on MINIX 3. In order to stress the lower layers, the cache layer was given a small (64MB) buffer cache. A sync call is made once every five seconds.

3.7.1 Performance evaluation

For performance evaluation, we used these macrobenchmarks: PostMark, altered to perform a sync call before the transactions phases, configured to perform 80,000 transactions on 40,000 files in 10 directories, with file sizes between 4KB and 28KB, using 4KB I/O operations; FileBench File Server, altered to use the same randomly chosen random seed for each round of experiments, configured with 10,000 files at an average of 20 files per directory; FileBench Web Server, altered to pick files using a Zipf distribution pattern in order to introduce locality, configured with 25,000 files with an average of 20 per directory; and finally, an OpenSSH build test which unpacks, configures and builds OpenSSH.

TwinFS

We started by evaluating the performance of TwinFS, on a single disk, initially without a resynchronization log. Application data was unprotected (not twinned). We compared various TwinFS configurations to the original PhysFS file store implementation. We did not succeed in time to get a journaling file store to perform well enough for a head-on comparison to another crash-consistent layout.

In early experiments, TwinFS kept outperforming PhysFS. This turned out to be due to TwinFS' delayed block freeing, which resulted in more favorable block allocation patterns. We modified PhysFS to perform the same delayed freeing; we refer to this version as "PhysFS+df."

For TwinFS, we varied the hardcoded offset between the left and right twin of all pairs. A twin offset of 1 block means the twins are adjacent on the device. We show the results for offsets of 4, 8, 16, and 32 blocks; other twin offsets did not result in overall more favorable results. For comparison purposes, we also tested a twin offset of 0, causing all blocks to be updated in-place. This effectively reduces TwinFS to PhysFS+df, with one major difference: TwinFS issues device write cache flushes when writing the root block.

Table 3.1 shows the median performance result out of five runs for each configuration and benchmark. For all tests, the cache size was too small to contain the working set. This resulted in long run times and a significant amount of I/O. Surprisingly, in some tests, TwinFS-0 performed better than PhysFS+df. We confirmed that this is due entirely to the added write cache flush call–an oddity of the disk used.

Compared to the best of the crash-unsafe alternatives, TwinFS with a twin offset of 16 blocks yielded the overall best performance in our tests (with an overhead of 2–8%), although only by small margins compared to the other twin offsets. Microbenchmarks showed that performance degrades when otherwise contiguous metadata blocks use a mix of left and right twins, destroying contiguity. When this is not the case, performance goes up with larger twin offsets, because in that case the stretches of contiguous blocks are larger as well.

Data resynchronization

We used the same single-disk configuration and the same set of benchmarks to evaluate the overhead of the TwinFS resynchronization log. For each benchmark, we also measured the maximum amount of data that would have to be resynchronized after a system crash, if all the files were stored with redundancy.

The resulting performance numbers are shown in Table. 3.2. The extra run-time overhead was negligible. Also, the worst-case amount of data to resynchronize in case of a crash is very small in all tests–we confirmed that the resulting resync times are negligible (sub-second) even if all files were mirrored on two disks. Finally (not shown), none of the benchmarks ended up writing more than a handful of resync log blocks between any two checkpoints, suggesting that TwinFS requires only a small

Benchmark	T-4	T-8	T-16	T-32	Peak
PostMark	1197	1161	1146	1153	1663 KB
File Server	342.61	345.74	345.83	342.04	1327 KB
Web Server	530.25	536.86	546.52	539.03	4 KB
OpenSSH	632.23	632.53	631.90	632.33	1266 KB

Table 3.2: The same benchmarks, now for TwinFS with the resync log enabled. Also shown is the worst-case size of the user data to resynchronize if all data were stored with redundancy.

log area. Note that in our benchmarks, most in-place overwrites were the result of appending data to log files. Web Server only appends block-aligned chunks to its log, resulting in almost no data being overwritten.

Overall, the **prewrite** solution allowed for proper aggregation of checksum records. However, it proved not to be ideal, since it complicates parity precomputation for parity-striped files. We now believe that a better solution is to let the file stores cache small numbers of data blocks for short times.

The cache log

Next, we tested the overhead of the cache log, both in performance overhead and in resource usage. We configured Loris to mirror all files on two disks, using TwinFS with a twin offset of 16 blocks (and, although unused, a resynchronization log) on both mirrors. We did not bound the cache log size; rather, we measured the maximum amount of memory needed for the log between any two checkpoints. To keep the comparison fair, we do not let the cache save on I/O by reusing (meta)data stored in the in-memory log if it already evicted the primary copy; otherwise, this would speed up the performance due to a bigger effective total cache size. Table 3.3 shows the results.

As shown, the mirroring case added little overhead to the single-disk case. The cache log added almost no visible overhead on top of that. The memory usage for the log was significant, but this is expected to get relatively smaller with larger cache sizes, since more data pages will be shared between the main cache and the log in that case.

3.7.2 Reliability evaluation

For the reliability evaluation, we used the same configuration as for the cache log test.

Kill tests

We tested both the TwinFS checkpointing system and the process crash recovery procedure at once, by killing the processes in the lower two Loris layers. Each kill was performed by injecting a trap instruction at the process program counter.

We repeatedly ran the OpenSSH benchmark, killing one of the processes at random intervals–once every twenty seconds on average.

We performed 10,701 kills this way. In all cases, the crash was hidden completely from the application layer, and the benchmark completed successfully. The median cache log replay time was 0.22 seconds; the maximum was 6.5 seconds. This depended only on the size of the log. However, the logical and physical layers have to refill their local caches from disk after each restart; this caused an overall benchmark performance degradation of up to 44%. Of course, we do not expect crashes to occur this often in practice. In 85 cases, the cache refrained from replaying its log due to a crash at the very end of a sync. In 88 cases, a TwinFS instance had to reload its penultimate checkpoint.

Targeted fault injection

In addition, we manually injected a number of less-trivial process faults, based on bugs we experienced in practice while developing Loris.

Stack overrun: In early Loris versions, thread stacks had no guard pages. If the stack for one thread's execution path was too small, part of the next thread's stack would be overwritten. That would cause a crash in the next thread, but only if that thread was active. This bug occasionally triggered in TwinFS with deep recursive parent block updates.

Heap corruption: In one case, a static array in the logical layer was too small for the maximum amount of data stored in it, and this would sometimes result in other variables on the heap to be overwritten. An assert would then go off when one of the corrupted fields was used afterwards.

Deadlock: A typical source of transient failures is threads contending for shared resources. Any programming errors in mutual exclusion code can cause race conditions. During TwinFS development, we ran into a case where multiple threads would concurrently try to acquire a large number of data buffers from a shared pool for a write operation, resulting in buffer exhaustion and a deadlock between the threads. This would trigger call timeouts after a while.

We simulated these and 12 other comparable bugs in the latest Loris version. Each bug eventually triggered, and caused a process crash. In all cases, the recovery mechanism performed successful recovery, and the system kept running.

3.8 Related work

We list the most directly related work on system and process crash recovery.

3.8.1 System crash recovery

TwinFS can be described as a hierarchical version of doublefs [61], or a selective copy-on-write file store where each protected block has two preallocated copies.

Benchmark	No cache log	With cache log	Memory usage
PostMark	1132	1150	170 MB
File Server	337.79	337.70	192 MB
Web Server	531.85	540.60	112 MB
OpenSSH	635.05	634.28	230 MB

Table 3.3: The same benchmarks, now with two TwinFS-16 instances (with resync log) and all files mirrored across these two instances. Also shown is the peak memory usage for the cache log.

As such, this layout shares several advantages with copy-on-write file systems: no need to write out metadata updates more than once, and no metadata recovery code. At the same time, TwinFS does not share some of their disadvantages: high fragmentation, cascaded metadata allocation, and difficulty to track free space. Finally, the implementation is fairly simple. It does however require more on-device space. Compared to doublefs, TwinFS uses the hierarchy to determine which block copy to use, eliminating the necessity to read in both copies on every read. Intra-device redundancy, like Stable Storage [90], could be added as an orthogonal feature.

We are not aware of work that involves global consistency across arbitrary local heterogeneous file/object stores. A similar problem can be found in fan-out stackable file systems that require their own metadata storage. An example is RAIF [75], which does not fully address this problem.

Several solutions have been proposed to limit the areas of resynchronization in traditional software RAID. Most comparable to our work is journal-guided resynchronization [34], which proposes extending existing file system journals with records for data resynchronization. Our checksum records add checksumming to this, allowing the resynchronization procedure to determine not only what to resynchronize, but also which of the redundantly stored copies (not) to use. The checksum log is similar to the hash log used for general data integrity in [133].

The write hole can be eliminated by never overwriting data, thus avoiding the need for resynchronization altogether. This approach is employed by for example ZFS [136].

3.8.2 Process crash recovery

Little research has focused on storage stack reliability in microkernel environments. Studies that do (e.g., [31, 126]), depart much more radically from the traditional storage stack model, leaving open the question whether a reasonably efficient POSIX-compliant system could be built on top.

Membrane [138] uses checkpoints and in-memory logging to recover from crashes in existing file systems on Linux. We use the same basic approach, but can offer stronger guarantees because of the microkernel-provided process isolation, allowing recovery from a broader class of failures. This does come at an extra performance

cost; ongoing research aims to reduce that cost significantly by exploiting multicore architectures. Compared to Membrane, we protect the rather complex equivalent of the RAID layer. On the other hand, Membrane protects the equivalent of our naming layer. By leaving out the naming layer in this work, we can make fewer assumptions about the implementation of the layers we do protect. For example, the file stores are not required to generate the same physical file ID upon retried create operations, and open deleted files are not a special case.

Toward the other end of the spectrum, Re-FUSE [139] can recover from crashes in a more diverse set of file systems, at the cost of making more assumptions about their behavior.

3.9 Conclusion and future work

In this paper, we have made a case for integrated support for recovery from system and process crashes in the Loris storage stack, using a single shared base: restoring global metadata consistency by means of checkpointing. On top of this, we have presented a fast and reliable approach to data resynchronization for recovery from system crashes, and recovery from transient process crashes in the lower two layers of the stack with relatively few assumptions. Our proof-of-concept implementation shows that these additions increase the overall reliability of our prototype, at a reasonable performance cost.

Dealing with process crashes in other layers in the stack is part of future work, as sketched in [149]. For the naming layer, this involves making sure that every VFS call immediately flushes its state changes down to the cache. As a result, the cache will always be consistent with respect to the naming layer's stack metadata. That in turn means the cache can perform a sync operation at any time, allowing the cache log's memory size threshold to be strictly enforced (see Sec. 3.6.2). The cache layer itself is deemed a stable point in the stack; still, we are working on a new crash recovery technique for the cache, even though we have to make stronger assumptions about the failures that can occur.

One unsolved problem in this work is efficient support for *fsync* and transactions. We intend to investigate the implications of adding support for those. This will likely result in exploring the other option laid out in Sec. 3.3.1.

4

Battling Bad Bits with Checksums in the Loris Page Cache

Abstract

In this paper, we aim to improve the reliability of a central part of the operating system storage stack: the page cache. We consider two reliability threats: memory errors, where bits in DRAM are flipped due to cosmic rays, and software bugs, where programming errors may ultimately result in data corruption and crashes. We argue that by making use of checksums, we can significantly reduce the probability that either threat results in any application-visible effects. In particular, we can use checksums to detect memory corruption as well as validate the integrity of the cache's internal state for recovery after a crash. We show that in many cases, we can avoid the overhead of computing checksums especially for these purposes. We implement our ideas in the Loris storage stack. Our analysis and evaluation show that our approach improves the overall reliability of the cache at relatively little added cost.

4.1 Introduction

Reliability of operating systems is important because a failure in the operating system can affect all running applications on the system. The storage stack is the part of the operating system that deals with maintaining the user's data. Reliability of the storage stack is particularly important, because a failure in its components has the potential to destroy the only copy of important user data. This is especially true for the page cache: this component caches file data, but also holds file changes that have already been written by applications but have not yet made it to permanent storage. In this paper, we take a look at the page cache in the light of two reliability threats: memory errors and software bugs.

Memory errors, in particular those caused by external factors such as cosmic rays, may affect application and operating system memory anywhere at any time. Machines without error-correcting memory hardware are fully exposed to such errors. The page cache typically uses all of free memory for caching purposes, and is therefore relatively likely to get hit. Ideally, we would like the cache to detect memory corruption with a high probability before it gets the chance to spread to applications.

Software bugs are a different, well-known source of reliability problems. Bugs may cause arbitrary behavior when triggered. Ideally, we would like to recover from the effects of software bugs in the cache, in an application-transparent way. However, recovery can succeed only if the necessary internal state of the cache can be recovered, and the difficulty is assessing that this state has not been corrupted as a result of the crash.

In this work, we claim that we can address both these problems at little extra cost, by making use of specific information present in the storage stack. Specifically, we reuse the checksums already employed by the storage stack to detect disk corruption. If the page cache is brought in the loop regarding these checksums, it can reuse them for runtime verification of cached pages against memory errors, and for integrity assessment of the cache's internal state after a crash. This allows the cache to catch memory corruption with a high probability, and to recover from a crash when possible–all completely transparent to the running applications. We implement our ideas in the Loris storage stack, which we developed in previous work [9]. Our evaluation shows that the two solutions are not independent, and in fact interact in an overall beneficial way.

The rest of the paper is organized as follows. In Sec. 4.2 we describe our Loris storage stack. In Sec. 4.3, we analyze the two reliability threats, argue that the cache is especially important in respect to the threats, and show that checksumming has the potential to help alleviate both. We then detail our approach to use checksums against memory errors (Sec. 4.4) and against software bugs (Sec. 4.5). Sec. 4.6 describes our implementation. In Sec. 4.7, we evaluate our solutions. We list related work in Sec. 4.8, and conclude in Sec. 4.9.

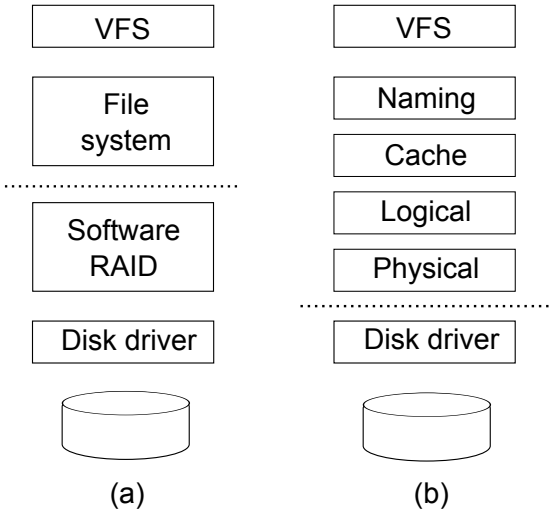


Figure 4.1: The figure shows (a) the layers of the traditional stack and (b) the new arrangement in Loris. The layers above the dotted line are file aware; those below are not.

4.2 Background: the Loris stack

Fig. 4.1a depicts the traditional operating system storage stack. Applications send requests to the Virtual File System (VFS) layer, which passes them to an actual file system. The file system operates on a single device; the RAID layer below may however transparently multiplex these operations across several devices for performance and redundancy purposes. Disk drivers are used to talk to the actual hardware.

In previous work, we have developed a new storage stack called Loris. It was formed by splitting up the traditional file system into three individual layers (a naming, a cache, and a layout layer), and swapping the layout layer with the traditional RAID layer, forming the physical and logical layers. The VFS and driver layers have been left as is. The result is depicted in Fig. 4.1b. This new stack has advantages in the areas of reliability, heterogeneity, and flexibility [9].

The four new layers communicate using files. Each file has a unique identifier and a set of associated attributes. The four layers support the following operations: *create*, *read*, *write*, *truncate*, *delete*, *getattr*, *setattr*, and *sync*.

At the bottom, the **physical** layer consists of one or more **file stores**. Each file store is responsible for one underlying device, and determines the layout on that device. It manages an independent set of *physical files* on its device, converting file operations from above to block operations below. Each file store has a small cache for the metadata of its layout. All file stores are required to implement a *parental checksumming* scheme in their layout [9]. As a result, the contents of the devices are fully covered by checksums, so that all forms of disk corruption are guaranteed to be detected.

The **logical** layer is the file-based equivalent of the traditional RAID layer. It exposes a *logical file* abstraction to the layers above, multiplexing operations on files across the file stores according to per-file RAID-like policies. Thus, each logical file has an associated policy and is made up of one or more physical files on different file stores. The logical layer stores this per-file information in a special *mapping* file, which is mirrored across all devices. It also implements RAID-like recovery for when an underlying file store reports a checksum or device failure. Our prototype implements file-based equivalents of RAID 0, 1, 4, and 5.

The **cache** layer implements a page cache, caching both file data pages and file attributes. It uses a large amount of memory for caching purposes, staging and evicting parts of files according to predefined policies. For performance reasons, it does not pass on incoming *create*, *setattr*, and *write* operations directly to lower layers; instead, it delays them to reduce request costs and to aggregate changes.

The **naming** layer converts POSIX operations to Loris file operations. It manages directories, which are treated as normal files by the lower layers. It uses Loris attributes to store the POSIX attributes of files.

We have implemented a prototype of the Loris stack on the MINIX 3 microkernel operating system. On this platform, all the layers of the stack (and all the file stores) are implemented as separate userspace processes.

4.3 The case for checksumming in the cache

In this section, we discuss two reliability issues: memory errors (Sec. 4.3.1) and software bugs (Sec. 4.3.2). We state why it is important to address these problems specifically in the Loris page cache, and we show that in both cases, we can significantly reduce the potential risks by making use of checksums.

4.3.1 Memory errors

The problem: Various studies have shown that dynamic random-access memory (DRAM) is susceptible to errors [107, 108, 169]. Such memory errors constitute arbitrary corruption in the system's main memory at unpredictable times and locations; unsurprisingly, software does not deal well with this. Memory errors are found to be responsible for a significant fraction of system failures in the field [123].

Memory errors are categorized as either soft or hard. Soft errors are transient changes in memory state (*bitflips*) caused by external factors; in particular, cosmic rays [169]. Soft memory errors are generally assumed to be distributed randomly in both space (i.e., affected memory location) and time. Various lab and field tests have suggested that soft errors are a serious problem [107, 108], and error rates in the 200-5000 FIT (Failures In Time) per Mbit range have been cited [95, 145]. More recent field studies found lower soft-error rates [95, 124].

Hard errors are caused by physical hardware faults, and may manifest themselves intermittently or permanently (*stuck bits*). The same recent field studies found hard

errors to be more common than soft errors [95, 124], although little is known about their exact cause. These studies observed that a relative minority of memory modules see the vast majority of errors, and that there is a strong correlation between the errors in both location and time [69, 124].

To counter the effects of memory errors, DRAM modules with error-correcting codes (ECC) have been developed. Typical ECC memory has single error correction and double error detection (SEC/DED) capability; the Chipkill memory family can also cope with whole-chip failures [32]. ECC memory can deal well with in particular soft errors. However, many computer systems are not equipped with ECC memory, primarily because of the added cost. This leaves them utterly exposed to memory errors. In a significant number of cases, these memory errors will end up causing serious damage to the running system [101]. Therefore, we believe that there is room for software approaches that can detect memory errors before they cause damage.

In this work, we consider only *soft* errors, on systems *without* ECC memory. All non-ECC memory is vulnerable to soft errors to some degree, and software solutions have the potential to overcome such problems. In contrast, not even SEC/DED memory can correct all hard errors [37, 69], and given that only a small subset of memory modules experiences such errors, the only real remedy for hard errors may thus be replacement of faulty modules. At the same time, we believe that any solution for soft errors can also help detect hard errors, although perhaps not as effectively.

Like hardware ECC memory, software approaches may use some form of redundancy (checksums, full copies, etc.) to detect and possibly correct memory corruption. However, such approaches are inherently imperfect: not every memory access can be guarded (e.g., device DMA), whereas any memory access can hit a bitflip. Software approaches can therefore only lower the probability of being affected by memory corruption, and more extensive forms of redundancy come at greater cost in terms of both performance and resource usage. Given the low probability of memory corruption, significant extra cost is typically not justified.

Therefore, we propose to start by significantly reducing the chance of undetected memory corruption while adding little extra performance and resource costs. We argue that the operating system page cache is the right place to start.

Why the cache: First of all, modern operating systems use all available memory (i.e., not used by applications) for caching purposes, so a large fraction of main memory is typically used by the cache for data pages. With a random spatial distribution of memory corruption, these data pages are therefore relatively likely to get hit by a soft error. Second, the operating system cache is shared by all applications. Memory corruption in the cache may affect the system integrity beyond just a single application. While this applies to all components of the operating system, measurements on our own systems revealed that the cache's pages typically use over 95% of all memory in use by the operating system. Third, in many cases we can recover from detected corruption without the overhead of keeping copies in memory: for clean data, there is already a valid copy on disk, so we can restore the data in the

cache from there.

The case for checksumming: In order to detect memory corruption in the cached data pages, we propose to use checksums. As we will show in Sec. 4.4, we can obtain checksums for the page data essentially for free, and that only leaves the verification aspect.

4.3.2 Software bugs

The problem: Another major source of reliability problems is software bugs. Bugs may cause arbitrary behavior during software execution. Previous research suggests that the number of bugs is a linear function of the number of lines of software source code [111] and that 0.5–6 bugs per thousand lines of code can be expected even in well-written software [57].

The operating system is of particular importance in this regard because a failure in the operating system may affect all running applications on the system. Microkernel-based operating systems allow failures to be contained, since most parts of the operating system are implemented as isolated system processes running in user mode. In that case, many software bugs in the operating system will manifest themselves as observable failures in the containing system process ("process crashes"). If the cause of the failure was transient–for example, a race condition–the system then *may* be able to recover transparently.

In the MINIX 3 operating system, crashed system processes can be restarted [62]. This is necessary but not sufficient for application-transparent recovery: most system processes have internal state, and after a restart, they must recover this state. Preferably, the state would be recovered from other system processes, so that the memory image of the crashed process need not be used for state recovery. However, this is not always possible.

Why the cache: The storage stack components of the operating system are directly responsible for storing user's data. Therefore, we believe that these parts deserve extra attention. In previous work, we have outlined how the system can survive process crashes in the Loris layers above and below the cache, without requiring reuse of internal state of the crashed processes [149]. This however leaves the cache layer, for which we will show such recovery is not possible.

The strict separation of the layers of the Loris stack into separate processes, and the process isolation provided by the underlying microkernel, ensure that the cache cannot be taken down as a result of crashes in the rest of the storage stack. However, the cache itself may crash. The current implementation of the Loris cache, including supporting library routines, is well in excess of 10,000 lines of code, and makes heavy use of nonpreemptive multithreading. It is therefore likely to contain dozens of bugs.

The case for checksumming: The cache is a crucial piece of the storage stack, and typically contains the only copy of significant amounts of application-generated state: dirty file data and other delayed file operations. If the cache crashes, this state

cannot be restored from external sources, and the cache can thus be recovered only if the crashed cache process's state is left in a proper condition. The main question is then how to assess this condition after a crash. To this end, we propose that the cache keep checksums of its crucial state during normal run time. As we will show in Sec. 4.5, we can generate checksums for a large part of this state at very little added cost.

4.4 Dealing with memory errors

The Loris file stores require checksums for all data in order to detect disk corruption. In this section, we discuss reusing those checksums for detecting memory errors in the cache. Since the file stores have to have a checksum for each block on disk, the cache can in principle get the checksums for all its clean pages for free. This obviates the need for *generation* of checksums specifically against memory corruption, and only leaves the *verification*.

Our goal is to reduce the window of vulnerability of undetected memory errors, while at the same time incurring little overhead. We focus exclusively on clean pages in this section, since clean pages may stay in the cache indefinitely, whereas dirty pages will be flushed to disk after (typically) at most 30 seconds, thus making them clean as well. We further discuss memory corruption in dirty pages in Sec. 4.5.

In this section, we first analyze whether the checksums used against disk corruption are usable against memory corruption at all (Sec. 4.4.1), and discuss how the cache can obtain on-disk checksums (Sec. 4.4.2). We then present a number of verification strategies (Sec. 4.4.3). Finally, we consider memory corruption in other memory of the cache and in other parts of the storage stack (Sec. 4.4.4).

4.4.1 Suitability of on-disk checksums

There may be significant computational overhead involved in verification of checksums. Thus, we first consider whether the on-disk checksums are the best choice for use against memory errors at all.

The Loris file stores use a checksum type from the family of Cyclic Redundancy Check (CRC) codes. Compared to simpler checksums such as exclusive OR (XOR), CRC codes are complex and traditionally implemented in software, and thus expensive in their use. However, it appears that an increasing number of platforms incorporates support for CRC in hardware [70, 135], closing the performance gap between XOR and CRC checksum computation.

At the same time, CRC codes are much stronger than XOR. XOR checksums have a Hamming Distance (HD) of 2, thus guaranteeing detection of one bit error only. The Loris file stores use the CRC-32C polynomial, which has a HD value of 4 for 5244 to 131072 bits [87], thus guaranteeing detection of up to three bit errors per block of the typical page and block size of 4096 bytes plus the checksum itself.

In addition, CRC codes are guaranteed to detect burst errors in length of up to the polynomial width.

This extra strength not only helps in detecting a wider range of on-disk errors, but also helps in detecting memory errors. A single cosmic ray may affect multiple adjacent memory cells at once [37], and cells within either the same row or column are likely to fall on the same page. The on-disk checksums are very likely to detect such corruption, whereas XOR-based checksums are not.

4.4.2 Propagation of checksums

We can involve the cache in the checksumming, by propagating checksums between the cache and the file stores. The most basic approach works as follows. Immediately before issuing a *write* operation to lower layers, the cache computes the checksum of each involved data page, and sends those checksums along with the write call. The file stores involved need not compute the checksum themselves any longer, and after the *write*, the cache will have the checksum for that page until the page is changed or evicted. In addition, when the cache issues a *read* operation for data, the file store always has to obtain and verify the checksum of the data anyway, in order to be able to detect and recover from disk corruption before the data reaches the cache. However, the file store now propagates up the verified checksum along with the data, so that the cache has the checksum from that point on at well.

4.4.3 Verification strategies

We now outline several strategies that the cache can use for checksum verification. They offer different tradeoffs between coverage against memory errors and overhead from verification. We note again that we can always recover after detection: all the pages involved are clean, and so a corrupted page can simply be read back from disk.

Background checker: On systems that see little overall storage activity, data may be held in the page cache for a long time. In general, the lifetime of data in the cache, and thus its vulnerability to errors, is potentially unbounded [167]. A single soft error may already get close to the limits of guaranteed detection of the CRC checksum; accumulation of multiple independent errors on a single page may be undetectable. To remedy this, the cache can slowly verify pages in the background. The expected time of accumulation is rather long even with high FIT/Mbit rates, but the cost of performing a slow background check is very low as well and puts a hard bound on the window of vulnerability. However, such background verification is independent from actual page accesses, and therefore ineffective at catching a single soft error before the affected page is accessed. Other strategies thus have to be considered in addition.

Check on every read: The cache could verify the checksum of a page upon every incoming *read* call involving that page. Combined with a background checker, this virtually closes the vulnerability window for clean pages. However, it comes at

a steep computational cost. Even for small subpage reads, the cache can only verify the checksum of the entire containing page. An application that reads from a file in small sequential chunks will force the cache to recheck the same pages very often with little time in between, resulting in significant CPU overhead and almost no gain.

Check on read after minimum last-check time: In order to alleviate this issue, the cache can prevent the same page from being rechecked too often. The cache could recheck the page's checksum only if the last check was at least a certain minimum time ago. This eliminates duplicate checks within that time frame, while ensuring that even frequently accessed pages will occasionally be rechecked.

Check on read after minimum last-use time: One could argue that there is no point for the cache to detect a memory error, if there is a high probability that that error has already been propagated to an application. From this perspective, there is no gain in rechecking a page that has been accessed recently before. Thus, a page's checksum could be checked only if there has been at least a certain amount of time since the last access. This approach leans even further towards the performance end of the spectrum, but may be less effective in practice: the application may not have actually hit the memory error on the earlier read (for example, due to small reads); also, different applications may access the same page.

Checking memory-mapped files: The options for verification of memory-mapped pages are limited, since the operating system is only involved in the first read from each page. Depending on the applications, memory-mapped I/O may not be all that common [54]. However, operating systems typically use memory-mapped files for shared libraries, and verification can thus help protect the integrity of important parts of running applications. The cache can use more generic techniques to reduce the vulnerability window of mapped pages. For example, it can perform more frequent background checks on these pages, or monitor them to trap after long times of no activity [35], at the extra cost and reduced accuracy of the polling-based monitoring.

4.4.4 Other memory

The cache's pool of data pages make up by far the largest fraction of memory used by the cache, and indeed by the entire Loris stack altogether. However, the cache's internal state comprises more than its data pages, and the remainder is worth at least some consideration as well.

Particularly noteworthy are the cache's data structures that describe the actual data pages, as every data page has to have such a corresponding data structure. However, this structure is much smaller than the page itself. For example, while not particularly optimized for size, the Loris page descriptor structure is currently 48 bytes, and thus about a factor 85 smaller than corresponding 4KB-page of data. The cache's data structures to track file objects and their attributes are similar in total size. Besides these, the cache has a long tail of smaller structures and global variables.

It may still be worthwhile to protect these parts of cache memory as well. Even

though they are less likely to be hit by soft errors, their corruption may affect the behavior of the cache itself–comparable to the effects of software bugs. More runtime checks on internal correctness can be added (for example with `assert` statements), but it is hard to reason about the coverage of this technique.

As an alternative, we attempted to implement checksumming for the internal data structures of the cache. We found that even with the simple design of the Loris cache prototype, manually adding support for checksumming of data structures added a prohibitive amount of complexity to it. While some parts were straightforward (e.g., linked-list macros and mutex operations were good places to start), a substantial amount of ground was left to cover with manual checksum update and verification statements.

We now believe that the solution lies in compiler support, in the form of programmer-guided (annotation-based) automatic checksumming of data structures, for example using the LLVM compiler framework. This would be similar to automatic approaches proposed for use against software bugs [48], but requires manual guidance so as not to automatically double-checksum the data pages as well. We believe the cache would be an interesting use case for such a technique, and we intend to work on this in future work.

Both asserts and semi-automatic checksumming would only enable error *detection*. In Sec. 4.5, we show that our proposed recovery system for software bugs can equally help recover from detected memory errors.

Finally, we note that the techniques presented in this entire section are also applicable to other layers of the Loris stack. For example, each file store has its own small metadata cache, and the checksum verification approaches for the page cache can be applied directly to this cache as well, although possibly with a different level of effectiveness. Any future semi-automatic checksumming solution may in fact be applied to large parts of the entire operating system, including the (small but crucial) microkernel.

4.5 Dealing with software bugs

In this section, we look at failures due to software bugs in the cache. Since MINIX 3 already provides several means of detecting misbehavior resulting from software bugs, we are concerned only with recovery. We propose a recovery approach that extends the checksum propagation from Sec. 4.4.2. In principle, it is otherwise fully independent from the memory errors solution from Sec. 4.4. However, we also show that there is a beneficial interaction between the two solutions if they are both used at the same time.

We first describe our assumptions regarding software bugs (Sec. 4.5.1). We then show how any recovery procedure needs a way to verify the integrity of the crashed cache's internal records of delayed operations, how this verification can be performed, and that checksums of all dirty pages are required for it (Sec. 4.5.2). We

discuss checksumming dirty pages next (Sec. 4.5.3), and then sketch the crash recovery procedure (Sec. 4.5.4). Finally, we show the benefits of checksumming dirty pages for memory error detection (Sec. 4.5.5).

4.5.1 Assumptions

Software bugs may cause arbitrary behavior. This includes scenarios where corrupted results escape detection and reach the application, in which case application-transparent recovery is impossible. Thus, we have to make assumptions about the errors we target.

First, we assume that if a software bug triggers in the cache, it will lead to a detectable failure; for example, a CPU exception, a failed assertion, or a bad interprocess call. In this case, the system can shut down the cache process, considering it to have *crashed*. Previous research on errors in operating systems suggests that a large majority of errors, if manifested in any way at all, indeed cause a detectable failure–silent failures are rare [51].

Second, we assume that the crash occurs within the execution of the requests that were active when the bug was triggered. This means that no bad results are propagated before the detection of the failure. Similarly, previous research has shown that fault propagation as the result of software bugs is relatively unlikely to occur [51, 77].

However, we do not assume that the failure was necessarily fail-stop, and want to attempt recovery even if (for example) arbitrary memory was overwritten within the cache. These assumptions are similar to those made in other contemporary work [31, 48, 93].

4.5.2 The Dirty State Store

As stated before, the cache typically has the only copy of many delayed file modification operations. This is the part of the cache state that cannot be recovered from elsewhere, and thus, the memory of the crashed cache process must be used to attempt recovery of such state. Thus, the first step for any recovery procedure is salvaging the delayed operations present within the cache at the time of the crash.

The cache supports three different types of delayed file modifications: *create*, *setattr*, and *write* (i.e., dirty pages). In order to assess the feasibility of recovery, the recovery procedure must be able to enumerate all delayed operations in the crashed cache process's memory image, and verify that they have not been corrupted as part of the crash.

To this end, we propose that the cache use an internal store to keep track of such dirty state. We call this store the Dirty State Store (DSS). It is a small and passive part of the cache, and exposes a very narrow API to the main cache code. Using this API, delayed operations can be added to the store as new application requests arrive, and removed from it as changes are flushed to disk. The store uses its own

(very basic) data structures to keep track of the operations, using a separate memory region.

However, this region is still part of the cache's address space, and any accidental overwrites from the main cache code could violate the integrity of the DSS. For this purpose, the DSS protects its data structures with checksums (XOR suffices for this). Thus, wild writes result in a checksum mismatch. The checksums need to be generated at runtime, but need never be verified unless the cache crashes.

We believe that the narrow and strictly checked API, the separate memory region, the checksumming of all parts involved, and the relative simplicity of the DSS allow us to put trust in the contents of the DSS if after a crash its checksums all match. Therefore, the recovery procedure can use the DSS memory to exhaustively enumerate and verify all delayed operations. The only requirement is that the DSS is kept up-to-date at all times, which means that its API must be used as part of each request handled by the cache.

While the file *create* and *setattr* operations can be duplicated in the DSS at little extra cost, keeping a copy of each dirty page in the DSS is not feasible: a substantial part of the cache's memory usage, and indeed of the memory in the system overall, may consist of dirty pages. Therefore, the DSS simply contains a pointer to the actual dirty page, along with a checksum of the contents of the page.

4.5.3 Checksumming dirty pages

The DSS thus requires an up-to-date checksum for each tracked (and thus, each dirty) page in cache. In terms of runtime overhead, checksumming of dirty pages dwarfs checksumming of the DSS data structures; this is where we return to interaction between in-memory and on-disk checksums.

In many cases, a page is modified only once and then flushed to disk sometime later. In such a case, performing the checksum computation upon the modification rather than the flush does not introduce any extra computational cost: the same checksum computation is simply performed a bit earlier. Thus, actual overhead is incurred only when the page is either modified again before the flush (due to another write operation), or it is discarded before the flush altogether (due to a truncate or delete operation).

Previous work suggests that cancelled writes (due to full-page overwrites, truncates, and deletes) are generally not dominant in workloads; for example, [18] reports a fraction of cancelled data bytes in the 4–27% range. This leaves subpage overwrites, to which we can apply two optimizations.

First of all, we find that a major source of subpage overwrites comes from file appends, thereby "overwriting" an unused part of the page. Many checksum types, including the CRC family, allow checksums to be computed incrementally. Therefore, instead of tracking only whole-page checksums, the cache can remember the size of the used part of each page (i.e., always the full page size, *except* for the last page of each file), and track the checksum of that part only–until the page is flushed.

For appends, the checksum can then be updated using only the new part of the page, thus avoiding checksum recomputation of the existing part.

Second, for small subpage overwrites, a CRC checksum can be updated for only the modified part of the page, by computing the checksum for just the old and the new parts with precomputed zero leads and trails, and XORing these partial checksums into the original checksum. Our initial tests show that this method can be more efficient than full page checksum recomputation.

4.5.4 Recovery procedure

With the DSS in place, we can now describe the overall crash recovery procedure. The recovery procedure is assumed to have full access to the memory image of the crashed cache process. Using the DSS and its checksums, the procedure starts by assessing whether all the delayed operations can be recovered from the crashed cache process. If this is possible, then the following steps are taken.

1. The recovered file modification operations are flushed down to lower layers. The result is that the cache is completely "clean" with respect to delayed operations.

2. All of the crashed cache's state is discarded, and the new instance of the cache starts with a clean slate.

3. The naming layer is notified that it should replay all ongoing requests. The Loris operations are all idempotent, so this is always safe to do, no matter what happened before the crash.

After that, the storage stack can resume normal operation, and applications will never notice that anything went wrong. However, if the recovery procedure finds that recovery of the delayed operations is not possible, or if replaying the ongoing requests repeatedly results in a crash of the cache, then recovery is not possible, and the system halts.

4.5.5 Consequences for memory errors

Our approach for dealing with software bugs yields two important positive effects in relation to memory errors.

The availability of checksums for dirty pages allows us to check these pages for memory errors as well, using the same verification techniques described in Sec. 4.4.3. In fact, in practice this simply means that we no longer have to make an exception for dirty pages. Moreover, since the checksums computed for the dirty pages are used directly as on-disk checksums, the result of an undetected memory error between the in-memory modification and flush to disk of a page, is that the memory error will be detected upon the next read from disk. In all these cases however, we can not recover the page, so whenever it is read later, the stack has to report an I/O

error to the caller. At the very least, this prevents corrupted data from reaching the application.

Additionally, as we noted before, memory errors may corrupt the cache's internal data structures, and even the program code. The result is that a memory error may cause the cache to crash. The crash recovery system can and will not make any distinction regarding the cause of the cache, and thus, the system will attempt to recover the cache in this case as well. As a result, the cache's primary data structures and code will be reset, wiping out any previous memory errors. Thus, the cache is able to survive crashes resulting from not only software bugs, but memory errors as well.

4.6 Implementation

We have implemented our ideas in the cache layer of our Loris prototype, on the MINIX 3 microkernel operating system. We briefly list what we implemented.

Memory errors: We have extended the Loris communication library to support propagation of checksums, and we have implemented basic generation and tracking of checksums in the cache layer. We have implemented the strategies described in Sec. 4.4.3 for detecting memory errors in cached pages, except those involving memory-mapped files, as MINIX 3 currently does not support those.

Software bugs: We have modified the cache to track checksums for dirty pages as well, including the append optimization (but not the partial-rechecksum optimization) described in Sec. 4.5.3. In addition, we have implemented the DSS. It offers a very narrow API of five calls (file create/setattr/flush, page write/flush) in under 200 lines of code, and the API calls were easy to add to the main cache code. MINIX 3 implements restarting processes after a crash, and optionally allows specified memory regions to survive process crashes. The restarted cache instance can perform the rest of the recovery procedure (as per Sec. 4.5.4) fully by itself, by verifying the checksums of the DSS and dirty pages, flushing down all dirty operations to lower layers, freeing the surviving memory, and requesting the naming layer to reissue any ongoing calls.

4.7 Evaluation

We evaluate our work on an Intel Xeon W3565 workstation, with 4GB of RAM, and a 500GB 7200RPM Seagate Barracuda SATA hard drive.

4.7.1 Microbenchmarks

We start with microbenchmarking the cost of basic *read* and *write* calls into the cache (without checksumming). We make calls directly from within the cache layer itself, rather than from an external source program: context switching is not optimized

Operation	Size (bytes)	Time (ns)
Read	1	802
”	4096	920
”	8192	1463
”	16386	2547
Write	1	799
”	4096	822
”	8192	1251
”	16384	2544

Table 4.1: Microbenchmarks for read and write calls of various sizes.

for performance on MINIX 3, and the overhead of context-switching through the other storage stack layers would otherwise reduce the relative cost of checksumming. Thus, the resulting relatively low times are worst case for us, and possibly more representative for other operating systems (we got very similar numbers in a userland benchmark on Linux). All pages involved are already cached in the cache layer, so no other layers are involved at all. The results are shown in Table 4.1. The (nanosecond) times are averages for a million iterations.

We also measure checksum computation for a single page, using the most optimized software and hardware CRC implementations that we could find. These measurements are shown in Table 4.2. They are representative for the additional cost of computing the checksum for a single page involved in a read and/or write operation. Thus, as can be seen, if checksumming of pages were added to the read and/or write calls, this would account for a substantial fraction of their time. We believe these overheads are not prohibitive: again, they are worst case, and applications are often not overly sensitive to such overheads [146]. However, the overheads are high enough to warrant exploring strategies that reduce the number of checksum operations.

4.7.2 Macrobenchmarks

Benchmarks: We perform further testing by means of macrobenchmarks. We use the following benchmarks in our experiments:

- PostMark (1.51), configured to perform 80K transactions on 40K files in 10 directories, with 4–28KB file sizes and 512B unbuffered I/O operations.
- FileBench (1.4.8.fsl.0.8) File Server, single-threaded, but otherwise with its default configuration, run for 30 minutes at once.
- FileBench Web Server, single-threaded, with 25,000 files, directory width 50, file size 32KB, and defaults otherwise, also with 30-minute runs. For Web Server, we use a modified FileBench version which accesses the files using a Zipf distribution rather than the default round-robin approach. With this

Operation	Size (bytes)	Time (ns)
CRC-32C in software [1]	4096	967
CRC-32C in hardware [71]	4096	205

Table 4.2: Microbenchmarks for page checksum computation.

(we believe, more realistic) distribution, we avoid that each cache page sees exactly the same access interval.

- An OpenSSH build test, which unpacks, configures and builds OpenSSH in a chroot environment.

The Loris page cache is given static size of 1GB of memory, and the tests were run on the first 32GB of the disk. All tests were run at least five times; average numbers are reported. For PostMark and FileBench, we consider the run phase only.

Table 4.3 shows statistics about the benchmark runs. The first column shows the percentage of *error consumption* [101]: the probability that if an error occurs anywhere in the cache's 1GB of memory any time during the benchmark run, the error will be read back, and thus propagate to an application or to permanent storage. These are the errors that matter; other errors simply end up being discarded. While the percentages may seem low, their upper bounds are the memory usage of the benchmark runs, reported in the second column. Thus, about half of the errors occurring in memory used by PostMark would be consumed, whereas this would be a tenth with the OpenSSH benchmark. The remaining columns of the table show read/write ratios (on a per-call and per-page basis) and the overall cache hit ratio.

Overhead and protection: We measure the overhead and memory error protection of our verification strategies these macrobenchmarks. For the overhead (O%), we count the sum of all page accesses by all incoming read and/or write calls, and per verification strategy, we report the percentage of page accesses that result in an *extra* checksum computation for that page. Thus, the higher the percentage, the higher the checksumming overhead. For the protection (P%), for each page we measure the time windows in which a memory error on that page would be propagated to the application or to permanent storage, and we sum the time windows of all pages together; per verification strategy, we measure the overall fraction of this total time where a memory error occurrence would be caught by the strategy. Thus, given a random memory error in the cache that would be consumed, the resulting percentage represents the probability that this error will be detected before reaching an application.

We list the results for the following verification strategies and time intervals: check on every read (ER); check on read after minimum last-check time (LC) of a MINIX 3 clock tick (1/60th of a second), 1 second, and 30 seconds; and, check on read after minimum last-use time (LU) with the same time thresholds. We have omitted the background checker results. As stated in Sec. 4.4.3, the background checker cannot stand on its own as a verification strategy, and our experiments show

that it indeed adds negligible protection for active workloads, unless configured to be prohibitively aggressive.

We first test our solution for memory errors from Sec. 4.4, which covers only clean pages. The results are shown in Table 4.4. The protection is below 100% even if every read is verified, because all techniques only cover clean pages, and all benchmarks involve at least some dirty data. The last-check and last-use strategies show that delaying checksumming by as little as one clock tick can significantly reduce the number of extra checksum checks, while keeping the protection at almost the same level. This is true especially for PostMark and OpenSSH, where the smaller (often subpage) I/O calls make rapid repeated page accesses common, thus resulting in overheads going down quickly. The per-tick strategies for File Server and Web Server maintain a high overhead because of the large (multipage) I/O granularity of these benchmarks. Higher time thresholds lower the overhead by much; the protection also decreases but remains substantial. There is little difference between the two strategies' results.

We repeat the same tests after adding our solution for software bugs from Sec. 4.5 on top of the memory error solution. This shows not only the extra overhead for checksumming upon write calls, but also the extra memory error protection resulting from having checksums of dirty pages. The results are shown in Table 4.5. We note again that we cannot recover from memory errors in dirty pages, but we do prevent corrupted data from reaching the application.

In this test, the overhead increases significantly because reads from dirty pages can now be checked as well, and the protection increases accordingly. These increases are shown by the new baseline, and reflected in the per-strategy numbers as well. Looking at the baseline, PostMark often reads back its own written data before it is flushed, and thus ends up with a high increase in protection. File Server has relatively the most writes, and thus ends up with a large increase in overhead– and, to a lesser extent, protection. Even with the every-read strategy, 100% protection is still not achieved: this is due to subpage writes causing entire pages to be rechecksummed, thereby losing the ability to detect previous memory errors in the unchanged page parts. We can solve this with the partial rechecksumming described in Sec. 4.5.3: this method preserves checksum errors across partial updates.

Benchmark	Err. cons.	Usage	Read call	Read page	Hit
PostMark	26%	54%	75%	75%	97%
File Server	18%	81%	37%	57%	41%
Web Server	23%	98%	93%	97%	91%
OpenSSH	10%	100%	83%	97%	98%

Table 4.3: Macrobenchmark statistics: error consumption, total memory usage, read percentage of all read/write calls, read percentage of read/write page accesses made as part of calls, and page cache hit ratio.

Benchmark	Baseline		ER		LC-tick		LC-1sec		LC-30sec		LU-tick		LU-1sec		LU-30sec	
	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%
PostMark	0	0	21	49	4	49	2	49	2	48	4	49	2	49	2	48
File Server	0	0	46	72	28	72	27	72	12	51	28	72	27	72	8	40
Web Server	0	0	97	96	80	96	42	95	13	74	80	96	36	95	8	66
OpenSSH	0	0	99	96	13	94	2	71	0	47	13	94	0	52	0	45

Table 4.4: Overhead and protection against memory corruption, using various verification strategies on clean pages.

Benchmark	Baseline		ER		LC-tick		LC-1sec		LC-30sec		LU-tick		LU-1sec		LU-30sec	
	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%	O%	P%
PostMark	4	40	97	91	52	91	8	91	6	88	52	91	7	91	6	88
File Server	41	20	88	99	70	99	69	99	52	76	70	99	69	99	49	66
Web Server	0	3	98	99	81	99	43	99	14	77	81	99	37	98	9	70
OpenSSH	0	3	99	99	13	98	2	74	0	50	13	98	0	55	0	48

Table 4.5: Overhead and protection against memory corruption, this time also checksumming dirty pages immediately.

Performance: The overall performance of the benchmarks is shown in Table 4.6. The numbers represent run times of the benchmarking phases, relative to the unmodified Loris implementation (thus, lower is better). The "clean only" columns represent Loris configurations that check only clean pages for memory errors, and correspond to the first three configurations in Table 4.4: the baseline (BL) that implements no memory check and thus only adds checksum propagation; a page check on every read (ER); and, a page check after at least one clock tick of not checking that page (LC-tick or LCt). The "clean and dirty" columns add checksumming on writes and the DSS keeping overhead, and thus correspond to the first three configurations of Table 4.5.

The baselines show that the checksum propagation has no overhead at all, and checksumming pages as they are written adds only little overhead. Checking checksums on every read has a clear performance impact (up to 9% for OpenSSH). However, the LC-tick strategy reduces the overhead to at most 1%. We believe that this overhead is acceptably low, especially given that LC-tick still offers strong protection against memory errors.

4.7.3 Fault injection

We evaluated the effectiveness of our DSS solution against software bugs by means of software fault injection, using the methodology described in [64]. We ran the OpenSSH benchmark 80 times, each time injecting a set of 100 random faults into the cache process at once at a random time during the benchmark. After each completed run, we checked the file system contents against those of a normal run to ensure that no corruption was propagated.

In 74 of the 80 cases, the cache crashed as a result of the fault injection, but was able to recover using the DSS. In all these cases, the benchmark completed and the final check passed. In the remaining six cases, other parts of the storage stack crashed because of deviating behavior of the cache. These cases violate our assumptions, but may still be recoverable if we harden these other layers. In no cases did the cache detect an internal checksum mismatch in the DSS or its dirty pages. Thus, the checksums had no added value in this experiment. We believe this is mainly because the OpenSSH benchmark is not write-intensive.

We then switched to the more write-intensive PostMark benchmark. We first

Benchmark	**Clean only**			**Clean and dirty**		
	BL	**ER**	**LCt**	**BL**	**ER**	**LCt**
PostMark	1.00	1.03	1.00	1.01	1.07	1.01
File Server	1.00	1.03	1.01	1.01	1.02	1.00
Web Server	1.00	1.00	1.01	1.01	1.01	1.01
OpenSSH	1.00	1.09	1.01	1.00	1.09	1.01

Table 4.6: Performance relative to unmodified Loris.

changed it to verify the results of all its system calls, so as to detect any propagated corruption. We then ran it 80 times, each time injecting 100 faults of the *destination* type. This fault type simulates corruption by altering the destination of random instructions [64]. This time, the cache crashed 79 times. In 75 cases, the cache recovered and the benchmark completed successfully. In the other four cases, the cache detected a DSS or page checksum mismatch, and decided that it could not recover. Without the DSS and page checksums, the restarted cache would have propagated corruption in these cases. Finally, in the 80th case, one of the other layers crashed.

4.8 Related work

We list the most directly related research on memory errors and software bugs.

4.8.1 Memory errors

We are not aware of any previous research that evaluates the use of disk checksums to counter memory corruption on a single system. A Lustre design document shows how servers send ZFS disk checksums along with file data to ensure network traffic integrity, noting that with approach, the clients will detect any memory corruption in the server cache as a side effect [137]. Zhang et al [167] study disk and memory corruption effects on ZFS, and find that neither ZFS nor ext2 deal well with local memory corruption; we build on their suggestions in this work.

Generic software memory corruption detection and recovery techniques have been proposed [35, 39, 127]. However, these approaches are unable to leverage specific knowledge about the data they operate on, and thus require higher checksum generation costs, offer less effective detection strategies, and/or require more runtime resources for eventual recovery. Nevertheless, they can still be applied to other parts of system and application memory.

4.8.2 Software bugs

We are not aware of any work specifically addressing recovery from bug-induced errors in the page cache. Again, more generic techniques can be applied. These are typically rollback based. For example, the Akeso system [93] tracks Linux kernel state changes on a per-request basis, committing the changes only at the end of the request. Compiler-based techniques are used to prevent arbitrary memory corruption. When applied to the page cache, this results in expensive copies of all dirty data; the authors show a substantial overhead in write-intensive workloads. A similar technique uses microkernels for better scalability [48], but has the same basic overheads.

CuriOS' Server State Regions (SSRs) address a similar problem by making client state available to a microkernel system server only during a request from that client

[31]. If the server crashes, only the active client is killed. We believe that SSRs are not well-suited for a page cache, as it inherently shares pages between clients.

The Rio file cache [27] protects its pages by remapping them read-only when executing kernel code outside the page cache routines. This approach does not help if the cache routines themselves contain bugs, nor does it help detect memory errors. Due to space constraints, we omit a large body of other work on software bugs, but we note that many approaches (e.g., language-based ones) can not deal with memory errors.

4.9 Conclusion

In this paper, we have shown that by using specific knowledge about the operation of the storage stack, we can effectively deal with certain reliability threats at a relatively low cost. We have addressed the threats of memory errors and software bugs in the page cache, and shown that there is a two-way interaction between the solutions.

Even though we have focused on the Loris storage stack in this work, we believe that the techniques presented here are sufficiently generic that they can be applied elsewhere: the techniques to detect memory errors can be applied in virtually any page cache that can be involved in on-disk checksums, and the techniques to recover from software bugs can be applied on any other microkernel.

5

Transaction-based Process Crash Recovery of File System Namespace Modules

Abstract

In this paper, we describe the emerging concept of namespace modules: operating system components that are responsible for constructing a hierarchical file system namespace based on one or more individual underlying file objects. We show that the likely presence of software bugs in such modules calls for the ability to recover from crashes, but that the current state of the art falls short of the desired behavior. We then introduce a crash recovery solution that is based on transactions, and detail the requirements for a system to implement this solution. We apply our solution to two different use cases: the primary namespace module for a storage stack, and an extension module that exposes the contents of scientific data files. Our evaluation shows that the transaction system has low overhead and significantly adds to the robustness of the namespace modules.

5.1 Introduction

Relatively recent developments have brought about a new concept in operating system storage stacks, namely *namespace modules*: software components that construct and manage a hierarchical file system namespace, using one or more file objects managed by an *object storage layer* below. Traditionally, namespace management has been an integral part of file systems, but there are two developments that have resulted in the isolation of such functionality into a separate module.

First, after the success of distributed storage stacks that separate metadata (and thus namespace) management from object storage for scalability [46], such architectures are now being introduced on single-system storage stacks as well, mainly to make up for inherent reliability problems with block-level RAID in traditional storage stacks [8].

Second, with user space file system frameworks such as FUSE [2], it has become relatively easy to write namespace modules that expose the inner structure of individual files using a hierarchical model. While such modules can be used for internal maintenance of such files, a recent case study [54] has shown by that current-day file formats can be highly complex, and suggests that important information for the storage stack is lost by storing these files as a single blob. Thus, there is a case to be made for use these namespace modules as the primary means of access to such complex individual files.

Due to the complexity resulting from (in particular) multithreading for high performance, and the large amount of code typically required for proper parsing and manipulation of complex file structures, both these types of namespace modules are likely to contain software bugs. If triggered, these bugs cause a runtime failure within the namespace module, typically with severely damaging consequences for both the running applications and the underlying storage. While namespace modules are often already isolated in user space processes (e.g., [2, 9]), this is only a first step toward dealing with failures, and we claim that current situation leaves several properties to be desired: 1) application-transparent recovery from transient crashes, 2) fine-grained request failure in the case of repeated failures, and 3) preservation of integrity of the underlying file objects.

In this paper, we present a crash recovery solution for namespace modules, based primarily on atomic transactions. In particular, we show that by using transactions between the namespace module and the object storage layer, we can not only provide recovery from fail-stop crashes with the three aforementioned properties, but also use semantic information in the transactions to minimize the runtime overhead.

We implement the transaction framework in our own storage stack, and apply it to two example namespace modules: the stack's primary POSIX namespace module, and a new extension module that allows the internal namespace of HDF5 scientific data files [58] to be mounted into the standard file system hierarchy. Our evaluation confirms that our prototype implementation has relatively low runtime overhead and allows for full recovery from large numbers of injected faults.

The rest of the paper is structured as follows. In Sec. 5.2, we describe the motivation for this work, elaborating on both the emergence of namespace modules as a concept, and the need to improve on the current situation with respect to crash recovery of such modules. In Sec. 5.3, we present the design of our transaction-based solution. In Sec. 5.4, describe the storage stack we use as test platform, the implementation of the transaction framework on this platform, the changes made to the original POSIX namespace module to make it recoverable, and the new HDF5 namespace module with recovery support. Sec. 5.5 presents the evaluation of our implementation, assessing both the resulting reliability improvement and the performance impact. In Sec. 5.6, we list related work. Finally, we conclude with future work in Sec. 5.7.

5.2 Motivation

In this section, we describe the motivation of our work: the emerging concept of namespace modules (Sec. 5.2.1) and the need for a better solution for recovery from bug-induced crashes of such namespace modules (Sec. 5.2.2).

5.2.1 Namespace modules as an emerging concept

Within the storage stack part of the operating system, one layer of functionality that appears to be increasingly common is what we call the *namespace layer*: a layer in the storage stack that, based on one or more underlying individual file objects and their contents, constructs a namespace view for use by applications. This namespace layer can typically be found directly under the operating system's Virtual File System (VFS) layer, where it exposes a hierarchical file system namespace that is accessible through a standardized interface. This interface is most commonly the POSIX application programming interface (API) [5], and thus, the hierarchy is made up of files, directories, links, and so on. Instances of the namespace layer, which we call **namespace modules**, thus translate file system requests coming from VFS, into individual file object operations on the lower layers of the storage stack. We identify two main types of namespace modules.

Primary namespace modules

The first type of namespace modules is emerging as a direct result of a shift in storage stack architectures. The traditional storage stack has a single file system layer which converts file system operations to block operations. The lower layers thus operate on a block basis, and this block interface mixes both file data and namespace metadata. Redundant storage across devices (RAID) is performed at the block level. This model is depicted in Fig. 5.1a. More recently, storage architectures have started to separate the management of the namespace and related file metadata, from the storage of actual file data. In this model, depicted in Fig. 5.1b, the lower layer

exposes an abstraction of individual *file objects*. We refer to this lower layer as the *object storage layer*. It typically implements redundant storage at the object level, making the translation to block operations only at the lowest (device) level. On top of the object storage layer, the namespace layer is responsible for constructing a file system namespace out of the collection of individual objects. We call instances of this layer **primary** namespace modules, as they are the primary managers of the underlying storage.

This general concept was introduced in the distributed storage world [46]. It has since been widely adopted in distributed storage systems (e.g., [15, 29, 160]), and has sparked the development and standardization of object-based storage devices [6]. The main reason for this architecture is scalability, as the decoupling management of metadata from the storage of data allows the object storage layer to be distributed across a large number of fully independent data storage nodes.

However, the object-based architecture is now also being introduced in storage stack designs for single systems, both in research (e.g., hFAD [125] and our own Loris storage stack [9]) and in the real world (e.g., ZFS [136]). Even though scalability is not a major concern on a single system, there are several other advantages of this new architecture. Most importantly, since cross-device redundancy is now implemented at the object level rather than the block level, this redundancy functionality can now make use of object information. This allows the storage stack to not only solve fundamental reliability problems present in block-level RAID, but

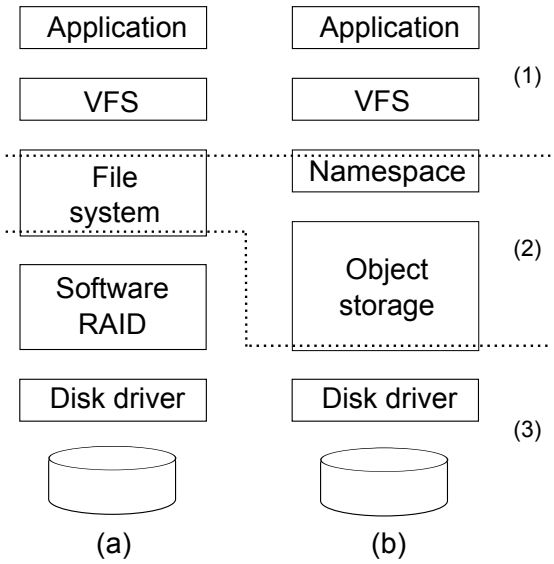


Figure 5.1: The figure shows the layers of (a) the traditional storage stack and (b) the object-based storage arrangement. The dotted lines delineate interface abstractions between layers: (1) the file system interface, (2) the object interface, not found in the traditional stack, and (3) the block interface.

also recover more files in case of a large number of concurrent device failures, and even store individual files with user-specified levels of redundancy [8]. In addition, the separation of the namespace layer allows it to be replaced easily and without affecting the underlying storage [148]. Furthermore, it allows for the introduction of several additional namespace modules that together manage the primary object storage space, thereby offering a variety of rich interfaces to the application in addition to the standard POSIX API [125].

Because of these advantages, we expect that the object-based storage architecture will see even more widespread adoption on single systems, and thus, that primary namespace modules will become more prevalent as well.

Extension namespace modules

At the same time, new user space file system frameworks such as FUSE [2] and PUFFS [3] have resulted in the emergence of a second type: **extension** namespace modules that expose the internal structure of a single file, by mapping its contents to the standard hierarchical file system model, allowing the file to be mounted into the system's file hierarchy. These modules thus "break open" the underlying file in a way that makes the file contents available to the end user and moreover, to any tool that uses the standard POSIX API.

Such modules are additions to the normal file system hierarchy, and typically loaded on demand. They can be written for any files of which the contents can be mapped to the standard namespace interface; for example, archive files, document files, and scientific data files. They can be used for inspection and maintenance of files generated by other applications, but also as the primary means of creation, manipulation, and usage of those files. As such, these namespace modules can serve to standardize access to these files between applications. In addition, a recent case study of complex file formats [54] suggests that storing complex files as binary blobs prevents the application from expressing its desires to the storage stack, for example to guarantee atomicity of operations on the file. This then forces the application to perform relatively expensive file system operations (such as full file copies and frequent `fsync` calls) to maintain such guarantees. Exposing the internal hierarchy of complex files through a namespace module would allow applications to better express their intentions to the operating system.

Even though stackable file systems [59] and Hurd translators [23] have been around for a long time, the new frameworks make development of namespace modules of this type accessible to a wide range of developers (and not just kernel developers), also on commonly used operating systems. In systems like FUSE however, the provider of the underlying file object is the entire storage stack, and the POSIX API is used for access to the file. We will show in Sec. 5.4 that differently structured storage stacks can allow extensions to talk directly to the object storage layer.

Hybrid cases

However, we believe that the two aforementioned types of namespace modules are in fact not so different, especially when considering virtualization. We are working toward a lightweight virtualization system that allows multiple namespace modules–one per virtual environment–to share a single object storage layer, thereby eliminating much of the redundancy found in contemporary virtualization systems [149]. These virtual environments contain individual applications, and thus have private sets of files and are created and destroyed on demand. The namespace modules in such virtual environments can thus be classified as either type.

5.2.2 The reliability problem

All these namespace modules can represent a substantial amount of error-prone code in the operating system layer. The primary namespace modules make up a crucial part of the storage stack. Unlike most parts of the object storage layer, the namespace layer has to process many application requests immediately, before any operation caching comes into play. The desire for high performance typically translates into extensive use of multithreading, which is a well-known source of reliability problems, especially so in file system code [97].

In contrast, extension namespace modules are not crucial to the storage stack itself, nor are they performance critical; however, they are expected to interpret complex file formats. Worse yet, since these namespace modules must be able to handles files they have not created themselves, they have to deal correctly with arbitrary contents in the underlying file. This requires either substantial amounts of newly written code, or the inclusion of an external parser library of which the quality is not always known. Previous studies have suggested that the number of bugs in a piece of software is largely a function of its source code size, reporting numbers in the range of 0.5–6 bugs per 1,000 lines of code [57, 111]. Thus, neither type of namespace module can be expected to be bug free.

With the possibility that software bugs can cause namespace modules to crash as a given, the first concern is then the stability of the operating system in general. This issue can be addressed by placing the namespace module in an isolated user-space process, as is done in microkernel operating systems as well as by user space frameworks such as FUSE. However, even in such systems, no attempts are made to recover the namespace module after it has crashed. At best, applications receive I/O errors for all requests that involve the crashed module. While this protects the operating system from damage extending beyond the boundaries of the module, we believe that this is insufficient, for three reasons.

First, the application is always exposed to the crash, even when the cause of the crash was transient. As indicated, such crashes could be the result of race conditions in a multithreaded environment. If the namespace module could be restarted after a crash, and the calls could be reissued, the transient crash would not occur again.

This would allow for crash recovery that is fully transparent to the applications.

Second, the entire namespace module is shut down after the crash, even if the crash was the result of a software bug in a specific code path of a request handler. For example, many file system bugs are found in error handling code [97], which triggers only exceptional conditions. In that case, repeating the call after recovery would thus crash the namespace module in the same way. However, if the system could recognize this case and fail only the corresponding application call for that particular request, all other requests (and thus, applications) can continue to make use of the namespace module.

Third, namespace modules typically translate single incoming application requests into multiple related object operations sent down to the object storage layer. For example, creation of a new file typically involves two steps: the creation of the new file object, and addition of the name record to its containing directory object. To maintain consistency of the underlying storage, such a set of operations is supposed to be atomic. However, it is possible that a namespace module sends down an incomplete subset of the operations, and then crashes. It will then leave the underlying storage in an inconsistent state. This also prevents subsequent recovery.

Thus, we argue that a proper crash recovery system for namespace modules can and should 1) recover the modules in an application-transparent way from transient failures, 2) fail only specific system calls in the case of repeated failures, and 3) never let fail-stop [14] crashes introduce corruption in the underlying storage. At the same time, namespace modules can be assumed to crash only in exceptional situations, and thus, such a system should have low overhead during normal runtime. In the rest of the paper, we will describe our approach to meeting these requirements.

5.3 Design

We will now present the design of a solution that is based on **transactions**. We start by stating our assumptions about the operating environment and failures (Sec. 5.3.1). We then show how transactions form the basis of our recovery system (Sec. 5.3.2), and the changes needed to the object storage layer (Sec. 5.3.3) and the VFS layer (Sec. 5.3.4). Finally, we list the requirements for namespace modules themselves (Sec. 5.3.5).

5.3.1 Assumptions

We assume that the namespace layer is positioned below the VFS layer and on top of an object storage layer, indeed as depicted in Fig. 5.1b. As such, the main purpose of a namespace module is to process typical requests coming in from the VFS layer [84]. The module handles each request by performing a number of operations on individual objects the storage layer below. Typical object operations are creation and deletion of an object; read, write, and truncate operations on the object; and,

retrieval and modification of attributes associated with the object. While the namespace module may use input from outside the storage stack (e.g., the current time of day), any modifications it makes as a result of a VFS request must involve stored objects only. In the case of extension namespace modules, only one underlying object is involved, and thus, no object creation or deletion will take place. We believe that these assumptions are sufficiently generic that they cover any single-system object-based storage architecture, as well as namespace modules in frameworks such as FUSE.

Software bugs may cause arbitrary behavior, including undetected propagation of corrupted results to the application or storage. Thus, we have to limit our solution to a subset of all possible failures. For failure isolation, we assume that the namespace module runs in a separate user space process, and that a basic infrastructure is in place to restart this process cleanly if it has crashed. Furthermore, we assume the following failure model. First, all failures are detectable, either by the module itself or the outside system. Previous research has suggested that silent failures are rare [51]. Second, failures are detected before corrupted results are propagated outside the process. Similarly, previous studies suggest that fault propagation is rare [51, 77]. This failure model covers more cases than fail-stop [14], because it also allows for wild memory overwrites as long as the overwritten memory is either not accessed by another process or (for example) protected by a checksum. In addition, the process isolation limits the impact of resource leaks. The failure model matches that of other contemporary work in this area [31, 48, 93, 139].

5.3.2 Transactions and recovery

Given these assumptions, it is easy to see how one can introduce a system of atomic transactions. In the course of processing a VFS request, all operations that modify underlying objects can be bundled into a single transaction, which is sent down to the object storage layer at the end of that request. Thus, each VFS request that spawns any object modifications at all, results in one such transaction. The request will only finish once either the transaction has been fully committed in the object storage layer, or it has been aborted due to an error, with no changes made to the underlying storage.

Such a transaction system has two effects for crash recovery. First, each transaction is atomic, and thus the object storage layer commits either the entire transaction or no part of it. In addition, each transaction makes a transition from one consistent state to another. Thus, there is no possibility that a crash of the namespace module causes inconsistency in the underlying storage. This covers the third point from Sec. 5.2.2, and paves the way for covering the first two points. Note that we do not discuss isolation between transactions in this work: while it is crucial that proper isolation is provided if multiple concurrent transactions can exist, a similar requirement exists in the original situation, and we believe that solutions for this are too implementation specific to be generalized.

Second, since the namespace layer must commit the transaction before the end of processing the VFS request, it can not have any pending changes in its memory that do not pertain to VFS requests that are currently ongoing. Thus, a crash of the namespace module will at most result in loss of pending state for the ongoing requests only. As a result, if the namespace module crashes and is restarted, the underlying storage layer is guaranteed to reflect all the changes made by previously completed VFS requests. For each currently ongoing VFS request, either none or all of its changes have been committed already–after all, it is possible that the namespace module crashes either before or after each corresponding transaction has been committed. As we will show in the next subsections, these conditions are sufficient to allow the namespace module to guarantee correct recovery from a crash, as long as some additional requirements are met.

5.3.3 Support in the object storage layer

The transactions are sent as atomic units to the layer below the namespace module: the object storage layer. In general, this layer is expected to provide a cache for object data and operations. Thus, while it must process each transaction right away, the transactions need not be persisted on a device immediately–a successful transaction requires only that the modifications have been taken in by the cache. In rare cases, the transaction may result in a failure at the object storage layer. For example, this layer may encounter a general out-of-space condition, or an integrity problem with data that needs to be read in to perform the transaction (e.g., for a partial write to a block). In that case, the entire transaction must be aborted, and an error must be returned to the namespace module.

We argue that due to the semantic information available in the transactions, the object storage layer need not implement full support for rollback of the operations in the transaction. All actions required to guarantee the successful execution of the transaction, such as checking available resources and reading in data blocks for partial writes, can be done before the actual transaction is executed. For example, write operations need not make expensive memory copies for rollback after a later failure. In multithreaded environments, some minimal rollback support may be needed, such as reserving space beforehand and canceling the reservation on failure. However, the actual operations will be deferred for performance reasons (e.g., delayed writes), and hence any device failure will be detected long after the corresponding operation has been acknowledged–this is the same in the traditional situation. Thus, by avoiding the need for rollback, the transaction support in the object storage layer has minimal overhead.

There is one exception to the above: the application may request *direct I/O*, in which case the object storage layer must complete the transaction only once the changes have been persisted on a device. The transaction must then fail atomically if any of its operations cannot be committed to the device immediately. In that case, full rollback support is required. However, the rollback overhead is likely to be

masked by the cost of the immediate device I/O. It should be noted that abstractions within the object storage layer may prevent rollback of some combinations of operations: while object create, write, and set-attribute operations can be undone, truncate and delete operations can not. For transactions that contain only one operation of the latter category, this operation can be performed last. We have not found any scenario where multiple such operations would be sent down in a single transaction by a namespace module.

5.3.4 Support in the VFS layer

The VFS layer must implement the necessary support for recovery after a namespace module has crashed and restarted. The first step is resynchronization of the namespace module to the current state of VFS, which reflects the result of all completed requests, and of none of the requests that were still ongoing to that module at the time of the crash. As part of this step, the recovery procedure in VFS must issue requests to reopen any other previously opened files and restore any mount points, for example.

As the second step, VFS must reissue all requests that were ongoing at the time of the crash. In particular, VFS should reissue them one by one, for two reasons: 1) for transient failures caused by race conditions, serial execution prevents such race conditions from happening again, and 2) for repeated failures caused by a bad implementation of a request handler, serial reexecution allows VFS to pinpoint the request that causes another crash, and abort just that request. This cleanly covers the first and second points from Sec. 5.2.2.

Especially in multithreaded environments, for some requests, the corresponding transaction may have already have been committed in the object storage layer before the crash. Thus, the namespace module needs a way to recognize whether a request has already been completed or not. To this end, at the very minimum, each VFS request must have a unique identifier that stays the same upon repetition of the request.

5.3.5 Requirements for namespace modules

With this infrastructure in place, a namespace module can recover from all failures in our failure model, as long as it adheres to the following four requirements.

First, the namespace module may not defer any modifications to objects until after finishing the corresponding VFS request. All operations must be part of the request's transaction. We believe that for namespace modules, this is not a prohibitive requirement. Unlike file systems, namespace modules generally sit on top of the storage stack's main cache, and thus, this will incur little to no extra device I/O.

Second, the namespace module must deal correctly with transaction failures reported by the object storage layer. In particular, if the module maintains its own read caches (e.g., for file attributes or directory data), then either these caches must

be updated (pessimistically) only after the transaction succeeds, or any (optimistic) modification before transaction commit must be rolled back. Since transaction failures in the object storage layer are expected to be highly exceptional (see Sec. 5.3.3), even more drastic approaches can be taken in this case, as we will show in Sec. 5.4.4.

Third, the namespace module must not expect that its modifications are visible to the lower layers before the transaction is committed. For example, it must not expect that an object read operation will reflect changes made by an earlier object write operation within the same request, as the write operation will be deferred as part of the transaction. This requirement can be fulfilled by the transaction framework, or by optimistic modification of local caches.

Fourth, the namespace module must handle repeated requests correctly. If a request's transaction was committed before a crash, but VFS never received the request's reply, then VFS will reissue the request after the restart. Idempotent requests can simply be performed again, but the same does not apply to nonidempotent requests. For example, a "create file" request from VFS that succeeded the first time, would fail with a file-exists error the second time. Thus, after a restart, the namespace module must recognize any such requests for which the transactions have already been committed. For atomicity reasons, this information must be saved as part of the transaction, and thus be maintained by the object storage layer. In our case studies (Sec. 5.4) we present an approach that requires no further extensions to the object storage layer.

In some cases, more than just the request identifier must be saved. For example, any results that had not yet been copied to VFS or the application, must be copied upon repeat. In addition, recall that VFS will restore a restarted namespace module to the state *before* the ongoing requests. Thus, any internal state changes resulting from repeated requests (e.g., adjusting open counts of files) must be replayed as well.

5.4 Implementation

We now describe the implementation of our design. We introduce our previously developed Loris storage stack (Sec. 5.4.1), the infrastructure changes we made to it to support transactions (Sec. 5.4.2), the modifications we made to make its primary namespace module restartable (Sec. 5.4.3), and a new restartable extension namespace module developed for this work (Sec. 5.4.4).

5.4.1 Background: the Loris storage stack

In previous work, we have come up with a new object-based operating system storage stack [9], by applying two modifications to the traditional storage stack. First, we split up the traditional file system into three separate layers: a namespace layer, a cache layer, and a device layout (*physical*) layer. Second, we swapped the physical layer with the traditional RAID layer (now the *logical* layer). We call this new

arrangement the Loris storage stack; it is shown in Fig. 5.2b. The VFS and device driver layers are unchanged. The four new layers communicate in terms of *objects*, which are file-like storage containers that consist of a unique identifier, a variable amount of byte data, and a set of associated attributes. The lower three layers can together be viewed as an object store. We will now briefly describe the four layers.

Analogous to what we described in Sec. 5.2.1, the **namespace layer** is responsible for the translation of VFS file system requests to file object operations. We currently have a single namespace module: the primary implementation of the POSIX namespace for our storage stack. This module stores both files and directories as objects; lower layers are not aware of the concept of directories. It uses object attributes to store POSIX attributes of files. It maintains two extra objects for its metadata: a bitmap to track in-use object identifiers, and a list of open deleted files needed for system crash recovery.

Below the namespace layer is the **cache layer**, where object data and attributes are cached, and various operations are deferred, for performance. This layer uses a large amount of system memory for this purpose, staging and evicting data as necessary. Below it, the **logical layer** implements the equivalent of the traditional RAID layer, but on a per-object basis. Each object has an associated policy that determines its RAID-like policy. For example, an object can be mirrored or striped across a number of devices. The logical layer constructs single "logical" objects out of one or more "physical" objects stored in the **physical layer**. This layer consists of physical modules that each manage the layout of one underlying device, thus translating object operations to block operations.

As we already sketched in Sec. 5.2.1, the new arrangement has several structural

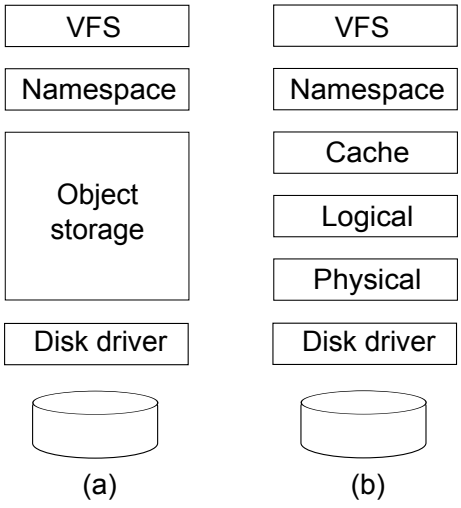


Figure 5.2: The figure shows (a) the generic object-based storage architecture from Fig. 5.1b, and (b) the layers of the Loris storage stack.

advantages, mainly for reliability and flexibility, as a result of the logical layer now operating at the object level. We have implemented our prototype of the Loris stack on the MINIX 3 microkernel operating system. In this environment, each module of each layer is a separate process running in user space.

5.4.2 Infrastructure changes

We implemented the transaction infrastructure described in Sec. 5.3.3 in the cache layer. Layers below the cache layer need not be aware of transactions. The transaction is fully maintained within the namespace module until it is committed, at which point it is sent down to the cache layer in its entirety. This saves on interprocess communication and hides local transaction aborts from the lower layer. We added a new "commit transaction" operation type to the communication protocol between the namespace layer and the cache layer, consisting of a set of one or more operations on objects. The cache ensures atomic execution of the transaction. Since our storage stack prototype does not yet support direct I/O, we did not add rollback support.

We also implemented the necessary changes in the VFS component, as per 5.3.4. We added identifiers to all VFS requests. Since VFS internally uses threads that block on requests to file systems, these identifiers consist of the combination of a thread number and a per-thread counter. Furthermore, MINIX 3 already supports crash detection and stateless restarts of system operating processes [63]. Thus, we only had to implement a recovery procedure in VFS: if it detects that a namespace module has been restarted, it first issues a number of requests that restore the previous state in the namespace module with respect to open files and mount points. It then instructs each thread waiting for a reply from that namespace module, to resend its request. This is done serially; if a repeated call triggers a new crash, VFS fails the corresponding application call with an I/O error. After reissuing all requests, normal operation resumes.

5.4.3 Case study: the POSIX namespace module

As our first case study, we modified the original POSIX namespace module of the Loris stack to incorporate proper transaction support. Internally, this namespace module consists of three sublayers: 1) a *dispatch sublayer*, which receives VFS requests, dispatches worker threads for them, and sends replies; 2) a *logic sublayer*, which implements the handlers for each of the VFS request types; and, 3) a *caching sublayer*, which consists of caches for its stored metadata (directory data, file attributes, in-use object identifiers, open deleted files). The adaptation of this module to the transaction system allowed us to assess the development effort of applying the transaction system to a module that has not been designed with transactions in mind. In particular, we wanted to know to which extent the "heart" of the module was affected: the logic sublayer. In addition, this use case allowed us to measure the

performance impact on a primary namespace module.

We started by adding basic support for transactions. Each worker thread now starts a transaction before invoking a request handler. All object modification operations performed by the handler are added to the current thread's transaction. Upon return from the handler, the transaction is committed, unless the request failed or the transaction is empty. The POSIX namespace module will never make any changes to objects as the result of a failing VFS request, and thus, if the request failed, the transaction is aborted.

We then changed the module to meet the additional requirements listed in Sec. 5.3.5. To satisfy the first requirement, we made the caching sublayer write-through, thus making its caches flush their changes immediately as part of the current transaction. To cover the second and third requirements, we decided to update these caches optimistically during the execution of request handlers. This way, we avoided changing the request handlers to track their own uncommitted changes. The only code change to the logic sublayer here was to allow rollback of changes to file attributes, since these were simply assignments to fields in a structure. We wrapped such assignments in macros that create a shadow record of the original value upon the first assignment in each request.

For the fourth requirement, we changed the namespace module to maintain records that contain the necessary information to skip repeated requests. We call such records *request recovery records*. Each record corresponds to a particular VFS request, and contains the request identifier, the reply fields expected by VFS for the request (e.g., an updated file size in the case of a "file write" request), and any file open count change that resulted from the request. We had to add a small number of calls to the logic sublayer to track the latter.

In order to store these records across requests, we changed the namespace module to create an extra object in the object store whenever it is started. The object is destroyed upon clean shutdown. Each request writes the corresponding request recovery record to this object as part of its transaction.

After a crash, the namespace module is restarted, and it will find that the object still exists. It then reloads the records from the object. Subsequently, while processing VFS requests, it compares each incoming request identifier to its records in memory. If there is match, the request is skipped: any file open count change in the record is reapplied, and a success code is sent back to VFS along with the stored reply fields in the record. Any memory copies to VFS or the application (e.g., the data for a "file read" request) are performed before the transaction is committed[1]; these need not be saved or repeated.

Maintaining request recovery records incurs little extra overhead. By using the VFS thread number embedded in the request identifier, we can aggressively recycle record entries in the object. As a result, the object is small in size and updated frequently, and thus expected to remain in the cache layer's (RAM) cache at all times.

[1]The transaction may still fail, but the POSIX standard does not require that application memory buffer contents be preserved upon call failure.

Therefore, updating a record as part of an existing transaction is cheap. Requests that end up in failure will not commit their transaction, and read-only operations will not create a transaction at all. Thus, no record needs to be saved in these two cases: the requests can safely be repeated. Therefore, maintaining these records never requires extra transactions.

In summary, fulfilling the first three requirements required changes to the caching sublayer only, except for small changes in the logic sublayer to be able to roll back updates. Fulfilling the fourth requirement required changes to the dispatch sublayer only, except for a few added calls to track file open count changes in the logic sublayer. Overall we did indeed not have to make substantial changes to the logic sublayer.

5.4.4 Case study: the HDF5 namespace module

For the second case study, we developed a new extension namespace module from scratch. This namespace module allows one to explore and manage the contents of a scientific data file, by mounting a representation of its internal hierarchy in the local system's general file hierarchy. While we do not expect our implementation to be used for the *generation* of such files, we believe that offering access to the contents of such a file through the normal POSIX API will help users in *maintenance* of such files, allowing them to use existing and familiar POSIX tools to not only explore the file, but also to make corrections to the file's internal organization.

For this module, we chose HDF5, a commonly used file format for scientific data [58]. HDF5 files are internally structured as filesystem-like hierarchies, with support for regular data files (*data spaces*), directories (*groups*), soft and hard links, an object attribute system much like POSIX extended attributes, and so on. The contents of the data spaces do not always map well to the standard byte-oriented POSIX `read` and `write` calls, but we expect such an extension module to be used for managing the hierarchy rather than the actual data.

We based our implementation on the official HDF5 open-source library implementation [58]. This library is single threaded, and thus, so is our namespace module. We added a relatively thin layer that maps the VFS requests to HDF5 operations. In addition, the namespace module intercepts all I/O calls made by the library, and converts them into Loris operations on the underlying object. The namespace module can be mounted by specifying a HDF5 file and a mount point, and then operates below VFS and on top of the cache layer, next to any other namespace module in the namespace layer.

After completing an initial version, we added basic transaction support, and made the module recoverable by implementing the requirements from Sec. 5.3.5. Satisfying the first requirement was not as straightforward as we hoped. While the library provides a function to flush its caches to the underlying file (`H5Fflush`), as it turned out, this function does not guarantee that the file can be reopened in read-write mode afterwards. Thus, we had to resort to making the library close and reopen

the underlying file between each two requests. To lessen the performance impact of the library's resulting constant file data reloads, we added a small read cache to the I/O handling sublayer of the namespace module. Again, optimistic updating of this cache fulfills the third requirement.

Meeting the second requirement was problematic at a more fundamental level. Since all intercepted I/O calls have to be deferred until transaction commit time, the I/O interception code has to report preliminary success to the library. If the transaction then fails in the cache layer, there is no longer a way to revert internal state changes in the library that resulted from the earlier perceived success. Given that transaction failures in the cache layer are rare and hard to deal with sensibly, we opted for a drastic solution that works in all cases: upon getting a transaction failure from the cache layer, the namespace module sends an error back to VFS, and then purposely crashes. As a result, the library will reload the pre-request state from the underlying object upon restart, and VFS will not repeat the request; operation can then continue.

We covered the fourth requirement with a similar request recovery record system as for the POSIX namespace module. We avoid creating a separate object in the lower storage layer, instead storing the information in a temporary data space within the HDF5 file. For this namespace module, the records contain some additional information. For example, the library may make changes to the underlying file even when it fails a particular call. Therefore, we also have to create a request recovery record for failing requests, and thus, each record has to contain the resulting error code as well.

In summary, this case study shows that it is feasible for an extension namespace module to meet the requirements for recovery, even when embedding a library of which the internals are largely unknown. We expect that our approach can be reused in many other extension namespace modules.

5.5 Evaluation

In this section, we evaluate our work. In Sec. 5.5.1, we evaluate the performance of both the the POSIX namespace module and the HDF5 module. In Sec. 5.5.2, we perform fault injection experiments to evaluate the robustness of both modules.

5.5.1 Performance

The POSIX namespace module

We started by subjecting the POSIX namespace module to microbenchmarks (omitted due to lack of space). Initial performance results were not impressive; this turned out to be the effect of file access time updates. These updates resulted in generation of a transaction for each (otherwise read-only) "file read" request, and this caused

Benchmark	Unit	Better if..	Original	No writeback	Transactions	Recoverable
MINIX 3 build	seconds	lower	703 (1.00)	701 (1.00)	698 (0.99)	698 (0.99)
OpenSSH build	seconds	lower	532 (1.00)	536 (1.01)	541 (1.02)	545 (1.02)
PostMark	trans./sec.	higher	825 (1.00)	776 (0.94)	823 (1.00)	825 (1.00)
File Server	IOPS	higher	1265 (1.00)	1244 (0.98)	1243 (0.98)	1279 (1.01)
Web Server (Zipf)	IOPS	higher	12915 (1.00)	12973 (1.00)	13044 (1.01)	13067 (1.01)

Table 5.1: Macrobenchmark performance of the POSIX namespace module, comparing four versions using five benchmarks, and showing both absolute and relative numbers.

high interprocess communication (IPC) overheads. We believe that maintaining accurate access times is not essential in most environments, and in all experiments we turned file access times updates off. With this change, all microbenchmarks showed low overheads.

We then ran a number of macrobenchmarks, on four different versions of the POSIX namespace module. First, the unmodified version forms our baseline for the benchmarks (*Original*). Second, we changed the module's caching sublayer to flush all object modifications down to the cache layer within the scope of the request (*No writeback*). Thus, this version shows the overhead of the extra operations resulting from the immediate flushing. Third, we added transaction tracking and rollback support to the namespace module (*Transactions*). These changes are expected to add some more overhead. However, the transaction system now bundles all modifying operations of each request into a single transaction, which is sent down as a unit to the cache layer. As a result, this version should have reduced IPC overhead. Fourth and finally, we added support for request recovery records, thus fulfilling all requirements to allow for crash recovery (*Recoverable*).

We used the following benchmarks and configurations: a MINIX 3 source compilation in a chroot environment; an OpenSSH build test which unpacks, configures, and builds OpenSSH, also in a chroot environment; PostMark, with 80K transactions on 40K files in 10 directories, 4–28KB file sizes, and 4K I/O sizes; FileBench File Server, with its default configuration, run for 30 minutes at once; and, FileBench Web Server, modified to access its files using a Zipf distribution in order to be more realistic, also run for 30 minutes.

The experiments were conducted on an Intel Core 2 Duo E8600 PC, with 4GB of RAM, and a 500GB 7200RPM Western Digital Caviar Blue (WD5000AKS) SATA hard drive, running MINIX 3.2.1. The tests were run with 1GB of cache memory in the cache layer, and on the first 32GB of the disk. For the PostMark and File Bench tests, we consider the run phase only. We report the average of at least ten runs.

The results are shown in Table 5.1. Compared to the original namespace module, the recoverable version has an overhead of 0–2% across the benchmarks, with some benchmarks even showing a small performance improvement. Removing writeback caching had the biggest impact on PostMark, but the transactions effectively canceled this out by reducing IPC overhead. Reduced IPC overheads also explain the other small performance improvements; MINIX 3 is not particularly optimized in this regard. As predicted, maintenance of request recovery records added no significant overhead in any of the benchmarks. Overall, we believe that the runtime overheads are sufficiently low for a process crash recovery system for the primary namespace module of storage stack.

The HDF5 namespace module

The HDF5 namespace module was written from scratch and primarily intended for human-driven maintenance. However, our basic support for reading and writing

Original	Flush	Reopen	Transactions	Recoverable
631 (1.00)	530 (0.83)	38 (0.06)	37 (0.06)	36 (0.06)

Table 5.2: Macrobenchmark performance of the HDF5 namespace module, comparing five versions using PostMark, showing both absolute and relative transactions-per-second numbers.

data spaces proved sufficient to run PostMark on it. Thus, we were able to get some performance measurements here as well. We tested five versions: the initial version (*Original*); a version that calls `H5Fflush` to flush the library caches to the underlying file after each request (*Flush*), which as we noted is not sufficient to retain read-write consistency; a version that reopens the underlying file between requests to flush all changes (*Reopen*); a version that adds transaction support to that (*Transactions*), and the final version that also adds request recovery records (*Recoverable*).

The results are shown in Table 5.2. As can be seen, constantly reopening the underlying file results in a serious performance penalty. Our tests show that almost all this time is spent by the library on creating and destroying internal data structures–it is simply not optimized for this kind of usage. If calling `H5Fflush` had been sufficient, this overhead would have been limited to about 17%. Thus, we conclude that while it is possible to meet recovery requirements in an extension namespace module even if it uses a preexisting library as is, if this library has not been optimized for these requirements, the resulting performance may suffer. We maintain that performance is not critical for an extension namespace module, unless it is intended as primary means of access to the file–in that case, it is worth optimizing any included library as well.

5.5.2 Reliability

We assessed the reliability of our implementation by performing a number of fault injection experiments on the POSIX and the HDF5 namespace modules. We injected faults in each module while a benchmark was running in a continuous loop.

For the POSIX namespace module, we used two benchmarks. The first is PostMark, which we modified to verify the results of all its calls. As part of this, we made it write known patterns in its `write` calls, and verify the data returned from its `read` calls accordingly. The second is the OpenSSH benchmark, with an added verification step for the compiled binaries at the end. For the HDF5 module, we used the same modified version of PostMark. In addition, since the OpenSSH benchmark expects more from the file system than the HDF5 module can offer (e.g., device nodes), we instead wrote and used a custom benchmark which performs a number of hierarchy manipulation operations in a loop.

We limited ourselves to fail-stop fault injection, as there is no easy way to inject faults that match exactly the failure model from Sec. 5.3.1. In order to maximize fault injection coverage, we injected fail-stop faults into the namespace module process in two different ways: 1) killing the process at random times by sending it a fatal

Module	Benchm.	"kill" injection			"swifi" injection		
		I	C	R	I	C	R
POSIX	PostMark	1500	1500	**1500**	1500	1500	**1500**
POSIX	OpenSSH	1500	1500	**1500**	1500	1500	**1500**
HDF5	PostMark	1500	1500	**1500**	1500	1500	**1500**
HDF5	Custom	1500	1500	**1500**	1500	1500	**1500**

Table 5.3: Fault injection results, showing for each namespace module, benchmark, and types of fault injection, the number of times fault injection took place (I), the number of crashes (C), and the number of successful recoveries (R).

signal ("kill"); 2) using a software fault injection tool to overwrite a limited, random set of CPU instructions in the process with instructions that generate an exception ("swifi"). While the latter eliminates skew introduced by process scheduling, the former adds more multithreading-like coverage of these (single-threaded) benchmark runs, as the module may now also be killed while it is in the middle of performing an operation. We note that without our changes, all fail-stop failures would be fatal to the namespace module.

For each of the combinations, we injected faults 1500 times. For the "swifi" fault injection, we injected 100 faults each time. The results are shown in Table 5.3. In all cases, *all* injections caused the namespace module to crash. More importantly, all crashes were followed by successful recovery, and none of the benchmarks were affected by the fault injection in any way. We believe that this is a good indication that our implementation works and can indeed achieve the intended reliability improvement.

5.6 Related work

The closest to our work is Re-FUSE [139], which provides recovery from fail-stop process crashes in FUSE file systems. It logs the system calls (and their results) made by the FUSE file system while processing each file system request. After a crash, the file system is restarted and the pending request is repeated. The file system is then expected to perform exactly the same system calls, for which Re-FUSE replays the original results–after completion, normal operation can resume. Like our solution, Re-FUSE requires that the file system defer no operations across requests. In contrast to our work, Re-FUSE also allows use of nonstorage resources like network connections. However, it requires strict determinism from the file system, and offers no guarantees in multithreaded environments. As stated, we believe that multithreading in particular is not only a requirement for high performance, but also one of the main sources of transient failures. In addition, Re-FUSE can not cleanly recover the underlying resources in the case of a repeating crash; under the same failure assumptions, our system prevents inconsistencies. Compared to both our work and Re-FUSE, other process recovery solutions for file systems either require more

resources and processing (e.g., Membrane [138]), or provide more invasive recovery (e.g., CuriOS [31]).

The use of transactions as the basis for recovery from operating system failures is not new (e.g., [48, 93]). While these approaches can recover from failures in larger parts of the operating system, we believe they have two main disadvantages: 1) they require extensive changes to the entire operating system, and 2) the generality of the solution makes it harder to apply domain-specific optimizations. For example, we stipulate that the high overheads on file write operations in Akeso [93] are due to extra memory copies necessary for its rollback system. Our system can mostly avoid this.

Other work has explored exposing transactions to applications (e.g., [115, 132]). While such systems add more overhead, it should be possible to combine them with our solution.

Previous work suggested *microrebooting* of isolated components to recover from fail-stop crashes [24]. Our work could be seen as an instance of this concept, although by focusing on the storage stack, we address different aspects of the problem.

ZFS implements a namespace module in the form of its ZFS Posix Layer (ZPL) [136]. It uses per-request transactions from this layer to ensure atomicity of its modifications in the lower layers. However, the ZPL is not a separate process, and we are not aware of work on process crash recovery of ZFS.

5.7 Conclusion and future work

In this work, we have described and evaluated a way to improve the reliability of a key component in the next generation of operating system storage stacks. There is however more work to be done in this context.

So far, we have taken the POSIX interface and thus the presence of a VFS layer as a given. In future work, we intend to focus on namespace modules that bypass VFS and expose an API directly to user applications, exploring the requirements for making such modules recoverable as well.

As sketched in Sec. 5.2.1, another future goal is to have multiple primary namespace modules in virtual environments, on top of a single shared object storage layer. In order to deal with hostile modules, we intend to implement on-the-fly verification that transactions retain the overall integrity of the storage system. This is similar to other recent work [41]. However, by working at the object level, we have more semantic information available to achieve this.

6

Towards a Flexible, Lightweight Virtualization Alternative

Abstract

In recent times, two virtualization approaches have become dominant: hardware-level and operating system-level virtualization. They differ by where they draw the *virtualization boundary* between the virtualizing and the virtualized part of the system, resulting in vastly different properties. We argue that these two approaches are extremes in a continuum, and that boundaries in between the extremes may combine several good properties of both. We propose abstractions to make up one such new virtualization boundary, which combines hardware-level flexibility with OS-level resource sharing. We implement and evaluate a first prototype.

6.1 Introduction

The concept of virtualization in computer systems has been around for a long time, but it has gained widespread adoption only in the last fifteen years. It is used to save on hardware and energy costs by consolidating multiple workloads onto a single system without compromising on isolation, as well as to create host-independent environments for users and applications.

In these recent times, two virtualization approaches have established themselves as dominant: hardware-level and operating system-level (OS-level) virtualization. The two are fundamentally different in where they draw the *virtualization boundary*: the abstraction level at which the virtualized part of the system is separated from the virtualizing infrastructure. The boundary for hardware-level virtualization is low in the system stack, at the machine hardware interface level. For OS-level virtualization it is relatively high, at the operating system application interface level. This boundary determines important properties of the virtualization system: the former is generally thought of as more flexible and better isolated; the latter as faster and more lightweight.

In this paper, we take a top-down look at the virtualization boundary. We argue that the two existing approaches are extremes in a continuum with a relatively unexplored yet promising middle ground. This middle ground offers the potential to combine several of the good properties from both sides. We propose a set of "mid-level" abstractions to form a new virtualization boundary, where the virtualizing infrastructure provides object-based storage, page caching and mapping, memory management, and scheduling, whereas any higher-level abstractions are implemented within each virtual environment. We argue that this approach offers a viable alternative, providing flexible, lightweight virtual environments for users and applications. We implement the core of our ideas in a microkernel-based prototype.

The rest of the paper is laid out as follows. In Sec. 6.2 we describe virtualization boundaries as a continuum. In Sec. 6.3 we propose and discuss a new alternative. We describe our prototype in Sec. 6.4 and its evaluation in Sec. 6.5. Sec. 6.6 lists related work. We conclude in Sec. 6.7.

6.2 Virtualization as a continuum

Reduced to its most basic form, any virtualization system consists of two parts. The **domains** are the environments being virtualized, with (at least) user applications in them. The **host** comprises all system components facilitating the virtualization of such domains. We refer to the abstraction level defining the separation between these parts as the **virtualization boundary**.

In this paper, we consider only virtualization in which unchanged applications can run using machine-native instructions (as opposed to e.g. Java [12] and NaCl [165]), and in which domains are considered untrusted. In this space, two approaches

have become dominant: hardware-level virtualization (Sec. 6.2.1) and OS-level virtualization (Sec. 6.2.2). We argue that these are extremes in a continuum of virtualization boundaries, in which new alternatives have the potential to combine good properties of both (Sec. 6.2.3).

6.2.1 Hardware-level virtualization

Hardware-level virtualization [19, 22, 49, 163] places the virtualization boundary as low in the system stack as practically feasible. A host layer (*virtual machine monitor* or *hypervisor*) provides its domains (*virtual machines*) with abstractions that are either equal to a real machine or very close to it: privileged CPU operations, memory page tables, virtual storage and network devices, etc. The low boundary allows a full software stack (OS and applications) to run inside a domain with minimal or no changes. The result is strong isolation, and full freedom for the OS in each domain to implement arbitrary high-level abstractions.

However, the OS adds to the domain's footprint, while typically exploiting only a part of its flexibility. Several fundamental abstractions are common across OSes: processes, storage caches, memory regions, etc. Reimplementing these in isolation leads to duplication and missed opportunities for global optimization. Only the host side can solve these issues, but the low boundary creates a *semantic gap* [26]: the host lacks necessary knowledge about the higher-level abstractions within the domains. Many ad-hoc techniques have been proposed to work around this gap [44, 73, 91, 103, 104, 158].

6.2.2 Operating system-level virtualization

In contrast, with OS-level virtualization [76, 110, 117, 131, 166], the operating system itself has been modified to be the virtualization host. The domains (*containers*) consist of application processes only–all system functionality is in the OS. Each domain gets a virtualized view of the OS resources: the file system hierarchy, process identifiers, network addresses, etc. Since the OS doubles as the host, there is no redundancy between domains and resources can be optimized globally. Thus, OS-level virtualization is relatively lightweight.

However, merging the host role into the OS has downsides as well. First, it eliminates all the flexibility found in hardware-level virtualization: the domains have to make do with the abstractions offered by the OS. Second, the merge removes an isolation boundary; failures and security problems in the OS may now affect the entire system rather than a single domain.

6.2.3 The case for new alternatives

The placement of the virtualization boundary in the software stack clearly has important consequences. However, we argue that the two described approaches are

extremes in a continuum. With the boundary as low in the software stack as possible, hardware-level virtualization represents one end of the spectrum. OS-level virtualization represents the other end, with the boundary as high as possible without affecting applications. That leaves a wide range of choices in between these extremes.

In a middle-ground approach, the host layer provides abstractions to its domains that are higher-level than those of hardware, and yet lower-level than those of OSes. Each domain then contains a *system layer* which uses those "mid-level" abstractions to construct the desired interface for its applications, as illustrated in Fig. 6.1. This way, we have the potential to reduce the footprint of the domains while retaining much of their flexibility and isolation. The only point that we must compromise on, is the ability to run existing operating systems.

6.3 A new virtualization design

In this section, we present the design of a new point in the virtualization continuum. We first state our design goals (Sec. 6.3.1). We then present the abstractions making up the new virtualization boundary (Sec. 6.3.2). Finally, we discuss the properties of our design (Sec. 6.3.3).

6.3.1 Design goals

Our goal is to establish a new set of abstractions implemented in the host system and exposed to the domains. Each domain's system layer may use those abstractions to construct an interface for its applications. In principle, the domain should be able to achieve binary compatibility with existing applications. With that constraint as a given, we set the following subgoals.

First, the abstraction level should be high enough to bridge the semantic gap.

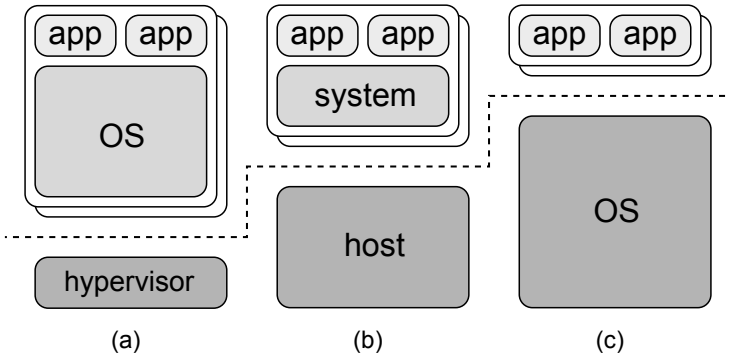


Figure 6.1: A schematic diagram of (a) hardware-level virtualization, (c) OS-level virtualization, and (b) our new alternative. The dashed line shows the virtualization boundary.

In particular, the new abstractions should allow for **lightweight domains**, mainly by reducing resource duplication. In many cases, the domains running on a single host will bear great similarity, because they implement the same application interface, and thus can share programs, libraries, and data, both in memory and on disk. The abstractions should make sharing of such resources a natural result rather than an afterthought, but with copy-on-write (CoW) semantics to retain full isolation. Bridging the semantic gap should also allow for other **global optimizations** generally found in OS-level virtualization only.

Second, the new abstractions should be sufficiently low-level to give the system layer in each domain substantial **flexibility** in implementing (or omitting) high-level abstractions for its applications. The host system should expose only abstractions that are well established as fundamental building blocks for operating systems and thus practically used in all domains.

Third, it should be possible to provide the new abstractions through a **minimal interface**. The virtualization boundary doubles as a boundary for fault and security isolation. A simple, narrow interface reduces complexity and security risks in the host system.

6.3.2 Abstractions

Based on these goals, we propose the following abstractions, starting with storage and going from there.

Object-level storage

For storage, hardware-level virtualization typically exposes a block-level *virtual disk* abstraction, while OS-level virtualization typically exposes a full-blown *file system* abstraction. We argue that neither is optimal.

Block-level storage again suffers from a semantic gap. Since algorithms at the block level lack filesystem-level information, block-level storage is plagued by fundamental reliability issues [88] and missed opportunities for optimization [33, 74], also in the context of virtualization [143]. In addition, virtual disks are intrinsically heavyweight: they are statically sized and the smallest unit of content sharing between domains is the whole virtual disk.

However, file systems represent the other end of the spectrum. They impose a specific naming structure and constraints, thus adding host complexity while taking away flexibility from the domain, in terms of semantics (e.g., POSIX vs Win32 file deletion), configuration (e.g., access time updates or not), and the ability to optimize metadata management for application needs [148].

Instead, we propose an object storage model [38, 46, 152], in which the host exposes an abstraction of *objects*: variable-size byte containers, each with a unique identifier and a set of associated attributes. An object can be created, deleted, read from, written to, truncated, and have its attributes be retrieved and manipulated. This

small set of operations makes up the core of the storage interface implemented by an **object store** in the host.

The object model imposes no structure *between* objects. It is left to the system layer in each domain to tie together its objects into a namespace. The basic approach is to use one object per file in the file system. Directories may be implemented as one object each, or with centralized objects [148], for example. With no hierarchy or metadata management defined at the object level, each domain may implement the file system abstractions appropriate for its user applications.

The details of storage management are left to the centralized object store. This store maintains and persists the global set of objects, on local or possibly networked storage. It gives each domain its own virtualized view of the set of objects, by keeping a per-domain *mapping* of local object identifiers to global objects.

The mapping facilitates storage space sharing similar to block-level CoW storage. The object store can map several per-domain identifiers to a single underlying object. Upon creation, a domain's initial mapping typically points to a set of preexisting objects. Such objects remain shared until a domain changes them, at which point it gets a private copy, thus retaining full isolation. As a result, the domains' object spaces will only ever diverge; to overcome this, the store can employ lightweight object-level deduplication to merge objects that become identical later. It can also implement finer-grained sharing and deduplication, for which it can use object-level hints.

Thus, overall, the object-based storage abstraction enables both per-domain metadata management flexibility and fine-grained cross-domain storage sharing. At the same time, the storage implementation details are fully confined to the host side, where the object-level information can be used to improve global decisions.

Object page caching and mapping

Next, we propose to extend the consolidation of shared storage objects to memory, in the form of a centralized **page cache**. Positioning the page cache in the host's object store yields several advantages. First, cache pages can be associated with their underlying global object, thus avoiding in-memory duplication of cached data between domains altogether. Second, a centralized page cache can employ global optimization strategies such as domain working set size estimation [73, 83, 98, 99] and exclusive caching on a second-level SSD cache.

With the page cache on the host side, the final step is to allow the cached pages to be CoW-mapped into domains, so as to let domains implement support for memory-mapped files. As a result, if multiple domains map pages from the same underlying global objects, these domains all end up with a copy-on-write mapping to the same physical page. This allows memory sharing of application and library code in particular.

Overall, the caching and sharing eliminates a substantial fraction of interdomain memory redundancy [25, 85] without the expense of explicit deduplication.

Address spaces, threads, and IPC

As supporting infrastructure, we propose that the host expose a number of microkernel-inspired abstractions: address spaces, threads of execution, and interprocess communication (IPC) [56, 142].

Thus, the host side becomes fully responsible for maintaining virtual memory and scheduling. It exposes an interface that allows for memory mapping, unmapping, granting, sharing, and copying, as well as creation and manipulation of threads. Centralized memory management not only simplifies the interface to our proposed memory mapping abstraction, but also facilitates centralized page fault and swap handling without the problem of double paging [50]. Centralized scheduling allows for global optimizations as well [82, 89, 141].

From the perspective of the host, each domain now consists of one or more processes making up its system layer, and a set of processes making up its application layer. In order to let these entities communicate, the host exposes IPC primitives, with access restricted as appropriate. The IPC primitives may also be used to implement additional required functionality for the domain system layer, such as timers and exceptions. We refrain from defining the exact primitives; such choices should be driven by low-level performance considerations and may be platform dependent.

Other abstractions

Physical resources typically require a driver and appropriate multiplexing functionality in the host. The main remaining resource is networking. Because of the state complexity of networking protocols, we believe the TCP/IP stack should be inside the domains. The abstractions exposed by the host can therefore simply be in terms of "send packet" and "receive packet," implemented by a host-side network packet multiplexer.

6.3.3 Properties

We now discuss several properties of our proposed virtualization approach and its implementations.

Flexibility and security

The system layer of each domain is free to implement any abstractions not exposed by the host. For a typical POSIX domain, this would include abstractions such as: process identifiers and hierarchies; signals; file descriptors; the file system hierarchy; pseudoterminals; sockets; network and pseudo file systems. A domain may implement the minimal subset needed by its applications to minimize the domain's footprint and its system layer's attack surface, or a set of experimental abstractions to cater to an esoteric application's needs, or a specific set of abstractions for compatibility with a legacy application, etc.

Compared to OS-level virtualization, our approach shares two advantages with hardware-level virtualization. First, a privileged user gets full administrative control over her domain, including the ability to load arbitrary extensions into the domain's system layer. Second, the host abstractions are available only to the domain's system layer, resulting in two-level security isolation: in order for an unprivileged attacker to escape from her domain, she would first have to compromise the domain's system layer, and then compromise the host system from there.

Performance versus subsystem isolation

Our design imposes no *internal* structure on the system layers on either side of the virtualization boundary. The host side may be implemented as a single kernel, or as a microkernel with several isolated user-mode subsystems. Independently, the system layer of each domain may be implemented as a single user-mode process, or as multiple isolated subsystems. On both sides this is a tradeoff between performance and fault isolation.

We leave open whether a monolithic implementation on both sides can achieve low-level performance on par with other virtualization architectures. For example, since the page cache is in the host system, an application has to go through its domain's system layer to the page cache in order to access cached data. In a straightforward implementation, this would several extra context switches compared to the other architectures. Future research will have to reveal to which extent any resulting overheads can be alleviated, for example with a small domain-local cache, memory mapping based approaches, or asynchronous operations [67, 106, 130].

For the host system, the resulting size of the trusted computing base (TCB) will be somewhere between that of hardware-level and OS-level virtualization. A microkernel implementation would allow for a minimal TCB for security-sensitive domains by isolating them from unneeded host subsystems, similar to Härtig et al [55].

Resource accounting and isolation

A proper virtualization system must give each domain a share of the system resources and prevent interference between domains [19, 52, 94, 131]. This problem is smaller when the host side provides fewer services: any resources used within a domain can easily be accounted to that domain, but the host must account its own resource usage to the appropriate domain explicitly. Our approach takes a middle ground: explicit accounting is needed for the object store and the memory manager, but not for higher-level resources. We expect to be able to leverage existing OS-level virtualization algorithms.

Checkpointing and migration

For common virtualization functionality such as checkpoint/restart and live migration [28, 110], our solution benefits from the middle ground. Compared to OS-level

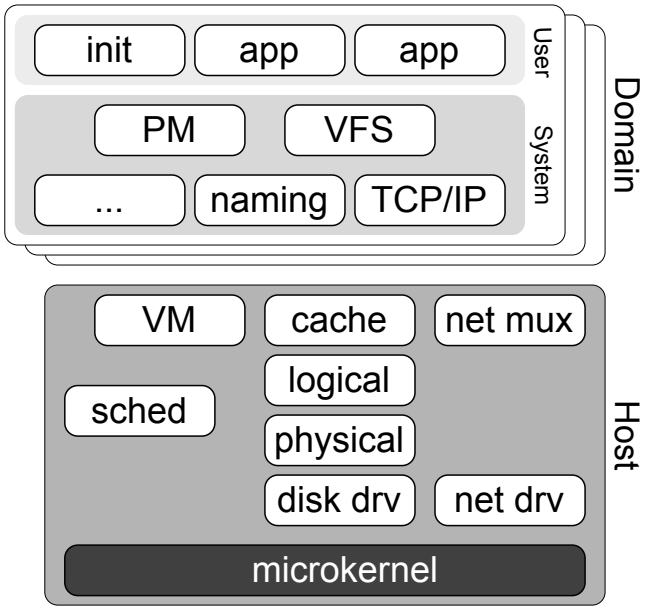


Figure 6.2: The components of our prototype.

virtualization, our host system provides fewer abstractions, and therefore less state to extract and restore explicitly. Compared to hardware-level virtualization, our central memory and storage management simplify the solution. The domains do not manage their own free memory, and any domain-specific pages in the page cache can be evicted at any time. Thus, the footprint of a domain can be minimized by the host at any time, obviating the need for ballooning [158].

6.4 Our prototype

We have built an initial prototype on the MINIX 3 microkernel operating system [142]. MINIX 3's microkernel implements a small set of privileged operations. This set includes low-level IPC facilities, which identify processes by global *endpoint* numbers. On top of the microkernel, POSIX abstractions are implemented in isolated user-mode processes called *services*. The most important services are: VM, the virtual memory service, which manages physical memory and creates page tables; SCHED, a scheduling policy service; PM, the process manager, which manages process IDs and signals; and VFS, the virtual file system service, which manages open file descriptors, working directories, and mounted file systems. PM and VFS provide the main POSIX abstractions to application processes, whereas VM and SCHED are not exposed to applications directly. Other services provide file systems, a TCP/IP stack, pseudoterminals, SysV IPC, etc. Hardware interaction is made possible with device driver services.

In prior work, we have developed a new object-based storage stack called Loris [9], which replaces the traditional file system and RAID layers positioned between VFS and disk drivers with four new, separate layers. Below VFS, the *naming* layer implements VFS requests, file naming, directories, and attributes. It stores files and directories as objects, using the object store implemented in the layers below: the *cache* layer, an object-based page cache; the *logical* layer, which adds RAID-like per-object redundancy; and, the *physical* layer, which manages the layout of underlying devices on a per-device basis and converts object operations to block operations. The object store uses the model and operations from Sec. 6.3.2.

We modified MINIX 3 and Loris to implement our virtualization concept. A virtualization boundary is drawn between the system services such that the host system consists of the microkernel, the VM and SCHED services, the lower three Loris layers, and all hardware driver services. Each domain has a system layer consisting of a private instance of PM, VFS, the Loris naming layer, and any of the other POSIX services as needed by its applications. The user application layer of the domain consists of a copy of the `init` user process and any actual application processes. The result is shown in Fig. 6.2.

Since endpoints were both global and hardcoded (e.g., PM has endpoint 0), we modified the kernel to virtualize endpoints using a mapping between global and domain-local endpoints. Thus, each PM instance has a domain-local endpoint of 0, but all PM instances have different global endpoints. The mapping also determines which host services are visible to each domain. We note that a microkernel with relativity of reference for IPC (e.g., object capabilities) would not require such a change.

We modified the Loris cache layer to support object-granular copy-on-write. To this end, we added per-domain mappings from domain-local to global object identifiers, as explained in Sec. 6.3.2, and global-object reference count tracking. An out-of-band interface is used to create and delete mappings for domains. We did not change the VM service in a similar way, as it already had support for copy-on-write memory.

We did change VM, SCHED, and the Loris cache to be domain-aware. These host services use a shared kernel page to look up any caller's domain by endpoint. In addition, we wrote a virtual LAN network multiplexer and a multiplexing console driver for the host system. We have not implemented support for resource isolation, checkpointing, migration, or storage deduplication at this time.

For implementation simplicity, there is an administrative domain which is able to set up other domains. Creating a domain consists of loading an initial object mapping into the Loris cache. Starting a domain consists of allocating a domain in the kernel, loading a bootstrapping subset of system services and `init`, mapping these processes into the new domain, and starting them.

Benchmark	Unit	Baseline	EV	EV,C
OpenSSH	seconds	393	1.01	1.02
AppLevel	seconds	621	1.01	1.01
FileServer	ops/sec	1178	0.98	0.98
WebServer	ops/sec	13717	0.97	0.96

Table 6.1: Macrobenchmark results.

6.5 Evaluation

For now, we believe that publishing microbenchmarks is not useful: neither MINIX 3 nor our new changes have been optimized for low-level performance. Instead, we provide statistics on memory usage and sharing, startup times, and storage macrobenchmark results. We used an Intel Core 2 Duo E8600 PC with 4 GB of RAM and a 500 GB 7200RPM Western Digital Caviar Blue SATA disk.

We tested our system with 250 full POSIX domains at once. Each domain consists of 13 service processes that make up the domain's system layer, plus `init`, a login daemon, and an OpenSSH daemon. The domains can all access (only) the following host services: VM, SCHED, the Loris cache layer, the virtual LAN driver, and the console multiplexing driver.

Each of these domains uses 3148 KB of *data* memory, which includes process heaps and stacks in the domain as well as host-side memory such as kernel process structures and page tables. In addition, the domains use 2944 KB of *text* memory, which is shared between all domains. Even with only around 3.7 GB of the machine's total memory being usable due to device mappings in 32-bit mode, the described setup leaves around 3.0 GB of memory available for additional applications and caching.

Creating and loading a CoW object mapping for a new domain, based on a subtree of the administrative domain's file tree (emulating `chroot`), takes 32 ms. Starting a domain to its login prompt, including OpenSSH, takes 53 ms. Shutting down a domain takes 4 ms.

We used storage macrobenchmarks to evaluate the performance impact of our main changes. Table 6.1 shows absolute performance numbers for unmodified MINIX 3, and relative numbers for the kernel's new endpoint virtualization (EV) and EV combined with CoW support in the Loris cache (EV,C). For the OpenSSH and AppLevel (MINIX 3) build benchmarks, lower is better. For FileBench's fileserver and webserver, higher is better. The exact configurations are described elsewhere [152].

The benchmarks show some overhead, caused mainly by new CPU cache and TLB misses. Optimizations should reduce these; all algorithms are constant-time.

6.6 Related Work

Roscoe et al [121] argue in favor of revisiting the hardware-level virtualization boundary. However, so far, proposals to expose higher-level abstractions [36, 114, 118, 164] generally keep the low-level boundary in order to support existing OSes. Decomposing monolithic OSes [45, 106] may eventually help port them to our architecture.

One exception is Zoochory [72], a proposal for a middle-ground split of the storage stack in virtualized environments. However, the proposal focuses on virtualization-unrelated advantages of rearranging the storage stack–ground we have covered in previous work as well [9, 11, 148]. They propose that the host system's storage interface be based on content-addressable storage (CAS); our object interface would allow but not enforce that the host side of the storage stack implement CAS. However, our redesign goes beyond the storage interface alone, and thus requires more invasive changes.

Our work shows similarity to several microkernel-based projects. L^4Linux uses task and memory region abstractions to separate virtualized processes from a virtualized Linux kernel [56]. Several L4-based projects combine microkernel features with virtualization without further changing the virtualization boundary [55, 60, 159]. The Hurd OS has "subhurds" [4]: virtual environments with long-term plans similar to ours [21]. Fluke [40] shares several goals with our approach, but its support for recursion requires high-level abstractions to be defined at the lowest level. New microkernel-based abstractions are also used for multicore scalability [86, 161].

We have previously presented an early sketch of our idea [149], focusing instead on its potential to improve reliability.

6.7 Conclusion

In this paper, we have established a new point in the spectrum of virtualization boundaries. Our alternative appears to have sufficiently interesting properties to warrant further exploration. However, with respect to virtualization architectures, we believe that the final word has not been said, and we encourage research into other alternatives in this design space.

7

Putting the Pieces Together: The Construction of a Reliable Virtualizing Object-Based Storage Stack

Abstract

The operating system storage stack is an important software component, but it faces several reliability threats. The research community has come up with many solutions to address individual parts of this reliability problem. However, when considering the bigger picture of constructing a highly reliable storage stack out of these individual solutions, new questions arise regarding the feasibility, complexity, reliability, and performance of such a combination. In previous works, we have designed a new storage stack called Loris, and developed several individual reliability improvements for it. In this work, we investigate the integration of these improvements with each other and with new functionality, in two steps. First, we add new virtualization extensions to Loris which challenge assumptions we made in our previous work on reliability. Second, we combine all our extensions to form a reliable, virtualizing storage stack. We evaluate the resulting stack in terms of performance and reliability.

7.1 Introduction

All computer systems are vulnerable to various faults, originating from both software and hardware. Such faults have the potential to completely subvert the operation of the system. Given that prevention and hardware protection are often infeasible or too costly, reliability can be improved using software-based isolation, detection, and recovery techniques. The storage component of the operating system is of particular interest in this regard: it is relied upon by all applications, it is highly complex, and it uses several resources which may fail in various ways. Since applications expect the storage stack to operate perfectly, unhandled failures in this stack may result in loss of user data.

The storage stack faces several important reliability threats. The most well-known threat is a *whole-system failure*, where the entire system goes down unexpectedly. The underlying storage hardware may experience a *storage device failure*, which may range from fail-stop failures to various forms of silent data corruption. Since the storage stack typically uses a large amount of available RAM for caching purposes, it is also vulnerable to *memory corruption* from sources such as cosmic rays. Finally, the complexity of the storage stack makes it susceptible to failures due to *software bugs*.

As a result, there has been much research on improving the reliability of the storage stack in various ways (e.g., [9, 17, 27, 31, 41, 53, 112, 138, 139, 150, 151, 152]). Such research typically focuses on a single reliability threat, limits itself to a single part of the storage stack, and often makes various assumptions about the component being protected. The result is extra reliability with low runtime overhead but with a limited scope. As such, while these efforts provide essential pieces towards making the storage stack more reliable, a top-down look at constructing a highly reliable storage stack raises new, important questions:

- Can a combination of individual techniques provide significant reliability coverage of the entire stack?
- How do these techniques hold up in the light of new functionality that may violate previous assumptions?
- What advantages, limitations, and complexity result from integrating various pieces with each other?
- What is the performance and reliability of such a combination of individual low-overhead techniques?

We believe that we are in a good position to provide initial answers to these questions. We have previously designed a storage stack called Loris, which is layered in a more modular way than the traditional storage stack [9]. In subsequent work, we have looked at the aforementioned reliability threats by considering individual layers of the stack in isolation [150, 151, 152]. By exploiting the specifics of

the design of these layers, we were able to provide individual solutions that provide strong reliability and add little overhead.

In this paper, we attempt to answer the above questions, in two steps. In recent work, we have presented a new approach to virtualization [154]; our first step in this paper consists of implementing support for this form of virtualization to Loris. In particular, we extend Loris with object virtualization, copy-on-write, and deduplication functionality, thereby forming **vLoris**. In addition to its intrinsic value, this first step serves two purposes. First, it improves reliability by placing the upper layers of the storage stack in their own failure domain. Second, the new functionality provides a good test case for the assumptions in our earlier work on reliability.

In the second step, we combine our earlier individual works on reliability with each other and with the new virtualization support, thus forming a virtualizing storage stack which is at least as reliable as the sum of the individual pieces. This new version of Loris, **rvLoris**, provides various levels of robustness against the four mentioned reliability threats across all its layers. However, given that we made strong assumptions about the layers in previous work, this integration effort required resolving several new issues. We describe the resulting changes that we had to make. Finally, we provide answers to the posed questions by evaluating the design, complexity, performance, and reliability of rvLoris.

The rest of the paper is laid out as follows. Sec. 7.2 provides a necessarily elaborate overview of our relevant previous work on Loris, reliability, and virtualization. In Sec. 7.3, we describe the virtualization extensions that make up vLoris. In Sec. 7.4, we describe our efforts to integrate all reliability and virtualization features to form rvLoris. In Sec. 7.5, we evaluate the performance and reliability of all our changes. Sec. 7.6 describes related work, and Sec. 7.7 concludes.

7.2 Background

In this section, we describe the prior work on which this paper builds: the Loris storage stack (Sec. 7.2.1), our reliability extensions to it (Sec. 7.2.2 to 7.2.4), and the role of Loris in our new approach to virtualization (Sec. 7.2.5).

7.2.1 The Loris storage stack

The basis of all our work is a new object-based storage stack called Loris [9]. This stack replaces the file system and software RAID layers of the traditional storage stack (Fig. 7.1a) with four new layers (Fig. 7.1b). These four layers communicate among each other in terms of *objects*: storage containers which are made up of a unique identifier, a variable amount of byte data, and a set of associated attributes. Each of the three lower layers exposes the following object operations to the layer above it: `create`, `delete`, `read`, `write`, `truncate`, `getattr`, `setattr`. In addition, a `sync` operation flushes all dirty state. Loris offers advantages in the areas of reliability, flexibility, and heterogeneity [9].

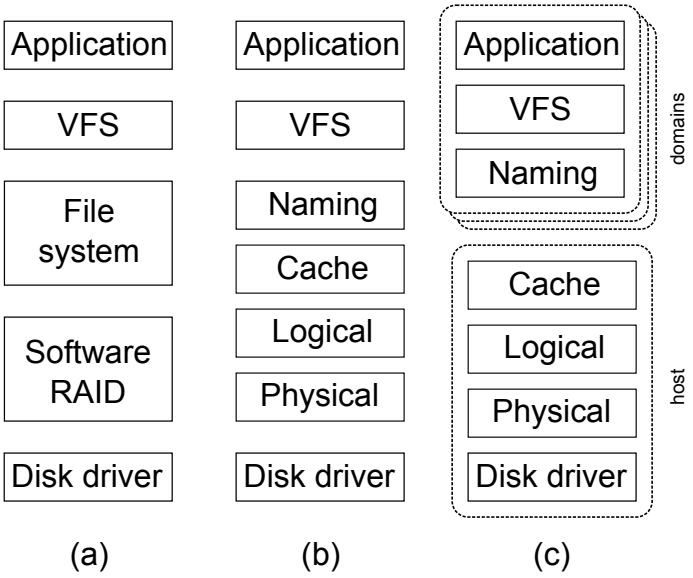


Figure 7.1: A schematic diagram of the layers of (a) the traditional storage stack, (b) the Loris storage stack, and (c) the Loris stack with virtualization.

At the bottom, the **physical** layer manages the layout of underlying storage devices, exposing a *physical object* abstraction to the layers above. The physical layer consists of one or more *modules*, each managing the layout of a single underlying device using a layout suitable for that device type. The physical modules are required to support parental checksumming to detect all forms of disk corruption [9, 88]. Our initial physical module, PhysFS, implements a layout based on the traditional UNIX file system, using *inodes* to store the objects with their attributes and data.

The **logical** layer adds support for RAID-like per-object redundancy. It exposes a *logical object* abstraction to the layers above. Each logical object has an individual RAID-like policy and is made up of one or more objects on different physical modules. For example, an object may be mirrored or striped across some or all devices. The logical layer maintains a mapping from each logical object to its policy and physical objects. It stores the mapping in a *metadata object*: an object which contains storage stack metadata and is mirrored across all devices. When a physical module reports a checksumming error, the logical layer uses the per-object redundancy to restore a known-good copy of the object. In the absence of redundancy, it still prevents propagation of corrupted data.

The **cache** layer uses available system memory to cache pages of object data. It also caches object attributes, and defers flushing dirty pages and object `create` and `setattr` operations for improved performance. Due to its limited role, the cache is by far the least complex layer of the stack.

Together, these lower three layers effectively implement an object store. On top

of these layers, the **naming** layer constructs a POSIX file system hierarchy out of the loose objects. It processes requests from the Virtual File System layer (VFS) and implements support for file naming, directories, and POSIX attributes. It stores files and directories as objects, and POSIX attributes as object attributes.

We have implemented the Loris storage stack in the MINIX 3 microkernel operating system [142]. As a result, all layers and modules of the storage stack are isolated user-mode processes, limited in their interprocess communication (IPC) in accordance with the principle of least authority. MINIX 3 has facilities to detect crashes in user-mode operating system processes and to start a clean copy of a failing process, although recovery of both state and ongoing requests is left to the process itself. The infrastructure by itself is sufficient to recover from failures in device drivers [62].

Summarizing, in terms of reliability, Loris can detect and recover from storage device failures by design, and MINIX 3 provides the necessary (but not sufficient) infrastructure to detect and recover from software failures ("crashes"). In subsequent, separate projects, we have further improved the robustness of Loris, and we describe these projects next.

7.2.2 Improved reliability in the physical and logical layers

In previous work [150], we argue that the storage stack should provide a unified infrastructure to recover both from whole-system failures and from crashes in the logical and physical module processes. For both failure types, recovery relies on the creation of consistent on-disk *recovery points*. A recovery point is created with the `sync` operation, which goes down the entire stack, during which all layers flush their dirty state to the layers below. Finally, at the physical layer, all modules create a recovery point on their devices. As proof of concept, we developed a new physical module, TwinFS, which supports such on-disk recovery points.

This infrastructure, when combined with cross-device data synchronization, guarantees recovery to the last recovery point after a whole-system failure. The same infrastructure allows for recovery of a crash in the lower two layers, when combined with an in-memory log in the cache layer.

The in-memory log records all modifying operations sent down to the lower layers since the last recovery point was created. It is cleared after a `sync` call. If one of the processes in the logical or physical layer crashes, all modules in this layer are restarted, and thus together restore to the last recovery point. After that, the cache layer replays the log, thereby bringing back the lower layers to the latest state. This approach provides transparent recovery with strong guarantees from transient crashes in the lower layers, and automatically covers any new features in these layers as well.

We exploit two properties of object storage to reduce logging complexity and overhead. First, since all objects are independent from each other, the log need not keep track of the order of operations between different objects. Second, operations that are later obsoleted (e.g., a `write` followed by a `truncate`) can be eliminated

from the log, so that the operations need not be replayed in chronological order.

Our evaluation shows that TwinFS adds a performance overhead of around 5–8 %. The log adds little performance overhead, but does require a large amount of extra memory, mainly to log pages that have been flushed and then evicted from the cache layer's (separate) main memory pool.

7.2.3 Improved reliability in the cache layer

As indicated, the Loris physical layer generates and verifies checksums of data and metadata blocks in order to detect disk corruption. We argue that checksums for data blocks should be propagated from and to the cache layer [151]. With such a facility in place, the cache layer can then use the checksums for two reliability purposes: detecting memory corruption and recovering itself from a crash.

First, the cache layer uses a large amount of system memory for data caching purposes. Therefore, there is a substantial chance that if a random DRAM bit flip occurs, it will affect a page in the cache. In order to detect such memory corruption before it is propagated, we change the cache layer to use the checksums for verification of pages right before application read calls. Recovery is often possible as well: corrupted clean pages can be restored from disk.

Second, crash recovery of the cache layer is difficult because this layer typically contains a substantial amount of state which cannot be recovered from elsewhere: delayed page writes, and `create` and `setattr` operations. Thus, if the cache process crashes, such dirty state must be recovered from the memory of the crashed instance of the process.

In order to ensure that the dirty state in the cache has not been corrupted as part of the crash, we change the cache layer to checksum the necessary data structures during runtime and verify them upon recovery. The state includes dirty pages; thus, each page is now checksummed right after being modified instead of when it is flushed to disk. All dirty state is tracked using a set of self-checksumming data structures, together called the Dirty State Store (DSS). During normal cache operation, the DSS forms a fully checksummed tree of all dirty state within the cache's memory. If the cache crashes and its recovery procedure detects a checksum mismatch in the DSS tree, the cache is not recovered, thus minimizing the risk that corrupted data reaches applications. An update of a page and its DSS checksum is not atomic; thus, this technique can recover from most but not all fail-stop failures.

For application-transparent crash recovery, the system must also deal with operations that were being processed by the cache at the time of the crash (as per Sec. 7.2.1). We require that either the naming layer or the low-level IPC system repeat pending operations after a restart of the cache. Since all Loris operations are idempotent, their repetition ensures a correct outcome even in the light of earlier partial processing.

The paper shows that the integration of the two techniques further strengthens both, and that the combination of hardware-implemented checksumming and simple

techniques can keep performance overhead down to around 1 % while providing reasonable detection and recovery guarantees.

7.2.4 Improved reliability in the naming layer

While the naming layer may be highly complex, at its core it simply translates each POSIX request from VFS into a number of Loris object operations to the cache layer. We exploit this fact to implement transparent recovery from naming-layer crashes [152], by making the layer stateless.

We change the naming layer to group all modifying operations spawned by a single VFS request into a transaction, which is then sent to the cache layer at the end of the request. The cache layer is responsible for processing each transaction as an atomic unit. As a result, the lower layers always contain a consistent state. If the naming process crashes, it will therefore also reload a consistent state upon recovery.

That leaves any VFS requests that were previously in flight. VFS reissues all those requests after a naming-layer restart; during runtime, the naming layer uses the transaction system to write markers to special files in order to determine after a crash which VFS requests were already processed before.

The cache layer places strict rules on the operations that may be part of a single transaction. For example, a transaction may contain only one `delete` operation: this operation can not be rolled back and must therefore be processed last. The cache further exploits the high-level nature of these transactions to limit rollback records to a minimum. In addition, the grouping of operations saves on low-level IPC; the resulting performance overhead is between −1 and 2 %.

7.2.5 A new approach to virtualization

In our most recent work [154], we present a new virtualization alternative which combines several good properties of virtual machines and operating system containers. The boundary between the virtualizing (*host*) and the virtualized (*domain*) parts of the system is placed such that the host can share and globally optimize resources across domains, while each domain has the freedom to implement a *system layer* which exposes any desired abstractions to its applications. As a result, our alternative is more lightweight than virtual machines and more flexible than operating system containers.

A key part of the design is a new form of storage virtualization: the host exposes an object store to the domains, and the system layer of each domain can use this object store to construct any file system abstraction as it sees fit for its applications. The central object store provides each domain with a private namespace of object identifiers; storage changes from one domain are never visible to another domain. However, the object store is free to share resources between domains, by mapping multiple domain-local objects to a single underlying global object using copy-on-write (CoW) semantics. Furthermore, if the object store implements a page cache,

such global objects can be shared not only on disk but also in memory. Since many domains are expected to use the same files in practice, this results in more efficient storage and memory usage in the common case.

The structure of Loris makes it a suitable basis to construct such a virtualizing object store. To this end, the storage stack is split in two (Fig. 7.1c). The cache, logical, physical, and driver layers become part of the host, together implementing the centralized object store. The VFS and naming layers become part of the system layer in the individual domains. Thus, each domain is expected to have its own instance of the naming layer, and all these naming modules make "host calls" to the cache layer to perform operations. All the storage stack modules remain separate processes, and the MINIX 3 microkernel implements domains by virtualizing IPC between groups of processes. As a result, the cache layer now receives a *domain identifier* along with each incoming operation. With this as a given, the cache layer must implement support for storage virtualization, which is the subject of the next section.

7.3 vLoris: support for virtualization

In this section, we describe the design and implementation of our new storage virtualization extensions to Loris. We call the result vLoris. We add object virtualization and copy-on-write support to the cache layer (Sec. 7.3.1). We reuse our previous work on transactions to support whole-system failure recovery (Sec. 7.3.2). We show that effective copy-on-write support requires attribute management to be moved into the cache layer (Sec. 7.3.3). Finally, we add object-level deduplication to the cache layer (Sec. 7.3.4).

7.3.1 Object virtualization and copy-on-write

We start by modifying the Loris cache layer to support object virtualization and object-granular copy-on-write. To this end, the cache layer maintains for each domain a *domain mapping object*: a metadata object (Sec. 7.2.1) which stores the mapping from the domain's local object identifiers to global object identifiers. When the cache gets a call from a naming module, it translates object identifiers in the call using the mapping for the naming module's owning domain. Thus, each domain can access only the objects in its mapping.

In order to support copy-on-write, we change the cache layer to keep a reference count for all global objects. These reference counts are stored in an additional metadata object. We use the domain mapping and reference count metadata objects to implement well-known copy-on-write semantics for the `create` and `delete` operations.

Whenever a domain modifies an object with a reference count greater than one, the cache layer makes a copy of the global object. In order to copy objects efficiently,

we introduce a new `copy` operation which is implemented in the logical and physical layers and used by the cache layer. Currently, the logical layer forwards the call to the appropriate physical modules, which make a full copy of the underlying object. A future implementation of this operation could implement copy-on-write support at a subobject granularity.

7.3.2 Transactions

An important part of virtualization is defining the interface between the host and the domains. In order to allow vLoris to recover from whole-system failures, its host/domain interface has to be changed, and thus, we consider this reliability aspect to be a part of the virtualization changes.

As described in Sec. 7.2.2, in order to establish a recovery point, all the Loris layers have to flush down their dirty state as part of a stack-wide `sync` call. This includes the naming layer. However, the naming layer is part of the domains, and the domains are untrusted from the point of view of the host system. Thus, they cannot be relied upon to cooperate in creating recovery points, since upcalls from the host system into the domains would introduce a dependency on untrusted components. Even though a naming module could only create inconsistencies for itself, such upcalls would at least need a timeout, and it would be very difficult to choose a reasonable value for this timeout. Avoiding upcalls is thus preferable.

We solve the problem by adopting the naming-layer transaction support described in Sec. 7.2.4. As a result, the naming layers need no longer be involved in establishing a recovery point at all: the transactions ensure that the cache layer always has a consistent state, and thus, the host system can create a new recovery point at any time and without the involvement of the domains. As a side benefit, this merges in most of the support for crash recovery of naming modules.

7.3.3 Attribute localization

In our work on naming-layer transactions, we disabled updating file access times for performance reasons. For this work, we added support for *lazy* access time updates in the naming module implementation. The naming module gathers access time updates for files, and periodically sends these down to the cache layer. While the naming layer is thus no longer stateless, deferring such updates does not introduce consistency violations. At most, a whole-system or process failure will cause some access time updates to get lost.

However, since each object is stored together with its attributes in the physical layer, the cache layer must copy any CoW-shared object whenever its attributes are changed. Access time updates make this design untenable: any read call on a file now causes sharing to be broken for that file.

This case makes clear that in general, effective storage sharing at object granularity requires that such sharing apply to the object's contents only, and not its

attributes. Thus, in our stack, we can no longer let the physical layer manage attributes. Instead, we "localize" attributes by storing them in the local mapping entries of the domain mapping objects, along with each global object identifier. The `getattr` and `setattr` operations are thus handled by the cache layer, and `setattr` no longer causes objects to be copied. The physical layer no longer stores attributes in its inodes.

7.3.4 Object-level deduplication

So far, the only way to increase the reference count of an existing object is to load a mapping for a new domain. Thus, after creation, a domain's storage resources will only ever diverge from the shared pool. There are however many scenarios in which multiple domains end up with the same objects (long) after the domains' creation, for example as a result of a software update being installed in several domains.

Thus, storage resources can be saved with object-level deduplication: a facility that finds and merges global objects with the same content and establishes copy-on-write mappings to the merged objects. We believe that deduplication at the object level is an acceptable tradeoff between yield and overhead [102], although as before, our object store can later be extended to implement subobject deduplication with no interface changes exposed to domains.

We implement support for such deduplication in the cache layer, since the object copy-on-write facility is implemented there as well. As we will now show, we use a weak (non-cryptographic) form of object content hashing to generate deduplication candidates with good accuracy and little overhead. We perform background content comparison on the candidate objects to achieve eventually-perfect deduplication.

As described in Sec. 7.2.1, Loris implements parental checksumming of blocks in the physical layer. We reuse these checksums to generate object hashes: the hash of an object is the exclusive-OR (XOR) of all its block checksums. The cache layer maintains an *index* which maps between global objects and their hashes. This index is stored in a metadata object, but also kept entirely in memory if possible.

Whenever the cache sends down an operation that changes the hash of an existing object–that is, a `write` or a `truncate` operation–the lower layers reply with a value representing the XOR of all changes to the object's block checksums. Since the physical modules visit all involved block pointers as part of the operation, getting the old block checksums from there is cheap. The cache XORs the reply value with the old object hash to obtain the new object hash, and modifies the index accordingly. While inserting the new hash value into the index, the cache checks whether any other objects have the same hash value, and if so, it registers the changed object as a candidate for deduplication.

At convenient times, the cache layer goes through the candidates list and compares each candidate against its potential matches, by doing a full comparison of the contents of the objects. We implemented a simple approach of reading and comparing potential matches at timed intervals. However, many optimizations are possible

here: comparing candidates before their pages are evicted, comparing block checksums before comparing actual contents, testing candidates only during times of low I/O activity, etcetera.

The cache layer has no reverse mapping from global to domain-local objects. Therefore, the candidates list contains pairs of domain identifiers and domain-local object identifiers. Upon actual deduplication, the candidate's domain mapping is changed to point to its match. The lack of a reverse mapping also implies that deduplication of objects within a single domain is automatic and cannot be prevented.

7.4 rvLoris: integration of reliability support

In the previous section, we already merged in support for naming-layer transactions. We now describe the changes necessary to merge our other reliability extensions with the new virtualization support as well as each other. Our goal is to maintain the same reliability guarantees as provided by the individual works. Since all the functionality comes together in the cache layer, all the new changes involve this layer in particular. We describe the changes that allow lower-layer restarts to work with virtualization (Sec. 7.4.1), and cache restarts to work with transactions (Sec. 7.4.2), object virtualization (Sec. 7.4.3), and object deduplication (Sec. 7.4.4). Finally, we discuss the results (Sec. 7.4.5).

7.4.1 Lower-layer restarts versus virtualization

In order to support lower-layer crash recovery for the logical and physical layers (Sec. 7.2.2) in the presence of the new virtualization, we have to make two modifications to the way the cache layer manages its in-memory log.

First, the logging facility was previously a separate submodule within the cache implementation: it maintained its own data structures and allocated its own memory to store logged data. This strict separation helped us in gathering research data about its implementation complexity and memory usage. However, the separate memory allocation was also a necessity: the only way to clear the log is to create a new recovery point, and since the cache layer itself could not initiate the creation of such points due to dirty state in the naming layer, the log was never allowed to run out of memory.

However, a real-world storage stack does not have infinite memory at its disposal; more realistically, such a log makes use of the same memory as the main cache [138]. Since the naming-layer transactions now allow the cache layer to create recovery points at any time (as per Sec. 7.3.2), there is no longer a need to allocate separate memory. Thus, we change our logging implementation to use the cache's main memory pool. This in fact simplifies our logging implementation: by using the main object and page data structures, the logging submodule no longer needs to manage duplicate data structures. In the new situation, cached pages that are in use by the

log may not be evicted until the next `sync`, but they may be freely read, modified (with new incoming `write` operations), and discarded (with `truncate` or `delete`).

Second, the addition of the new `copy` operation to the lower layers requires that the cache log include this operation as well. However, the `copy` operation violates the two assumptions that we previously relied on to keep the log simple: full independence between objects, and the ability to throw out any logged operations that have become obsolete. After all, the result of the copy depends on the state of the source object, and thus, a post-crash replay of this operation requires that the source object be restored in the same state first. The log must take this into account; e.g., an object `truncate` may not discard pages logged for an earlier `write` if the object was subject to a `copy` in the meantime.

The simplest solution is to establish a new on-disk recovery point after every `copy` call, thereby clearing the log. However, a system that hosts many domains can expect frequent copying, making this an expensive solution. Instead, we expand the cache logging facility with a partial ordering system, using hidden object and page data structures to log the precopy object state needed for replay. Basic dependency tracking allows log replay in a correct order: when a `copy` operation is logged, new data structures are created for the source object, and the old source object data structures are hidden and marked as dependency for the new ones.

7.4.2 Cache restarts versus transactions

As we have shown, the transactions system is used for both naming-layer reliability (Sec. 7.2.4) and, indirectly, for whole-system failure recovery for virtualization (Sec. 7.3.2). However, the addition of transaction processing to the cache layer has implications for the cache layer's crash recovery.

In our earlier work (Sec. 7.2.3), we required that pending operations be repeated after a cache-layer restart, and relied on idempotence to correct earlier partial completion of those operations. This no longer works with transactions. A single transaction may spawn multiple modifying operations to the lower layers, each of which may fail, in which case the entire transaction must be aborted. If some of these operations are issued successfully and then the cache crashes, and the transaction ends up being aborted upon repetition, the subsequent rollback will revert to an inconsistent state.

We solve this issue by adding rollback records for active transactions to the Dirty State Store (DSS). When the cache processes a transaction, it first creates a rollback record in a known (per-thread) location, along with a checksum generated of its contents. When the cache completes processing the transaction, it discards the record using a (checksummed) special value. After a crash in the cache, the cache's recovery procedure first verifies the checksums of all rollback records in the old memory image, and proceeds with recovery only if all are valid. It then issues a rollback for all those records, thus restoring the cache to a consistent state.

7.4.3 Cache restarts versus object virtualization

The cache layer's object virtualization adds a number of extra, risky steps for operation processing. Object creation, copying, and deletion actions all require changes to multiple objects: a domain mapping, the global reference counting metadata object, and often the global target object itself. With these extra steps, partial completion of even a single operation is no longer recoverable by repetition, as nonatomic metadata updates likely result in fatal inconsistency.

In order to ensure that the cache layer can still either recover correctly or detect internal corruption, we integrate the object virtualization steps into our transaction system. We split each of the creation, copying, and deletion actions into a prepare routine and a commit routine. We create a rollback routine for each action, which restores the old state regardless of any changes made in the meantime. We add new checks to ensure isolation between transactions, for example with respect to allocation of global IDs. Also, since it is common that two operations modify the same object within the same transaction, the prepare phase has to consider any actions already scheduled within the same transaction.

However, the cache cannot roll back `delete` operations already sent to lower layers. Thus, after a crash of the cache, a rollback of a delete action is not always possible. Therefore, we have to make one exception in the cache's internal consistency model: a domain-local object ID may temporarily point to a global object which has a nonzero reference count but does not exist in the lower layers. Since pending requests must be repeated after a cache restart (Sec. 7.2.3), such inconsistencies are always corrected quickly.

7.4.4 Cache restarts versus object deduplication

The addition of object deduplication to the cache layer requires three more extensions to its crash recovery system.

First, when the deduplication facility finds that two objects are identical, they are merged. Like the create, copy, and delete actions, such a merge action requires an atomic set of changes, and thus needs to be wrapped in a (purely cache-local) transaction. The action includes a `delete` on one of the two objects, making it similar to a delete action. However, a cache crash must not introduce a similar inconsistency for a merge action: the action was not started by a call from the naming layer, and thus may not be temporary. We therefore opt to let the merge rollback routine perform a roll-*forward* on the merge action. Thus, upon crash recovery, the merge will always be finished, thereby preventing inconsistency.

Second, the cache updates the deduplication index with a new checksum based on the reply to a `write` or `truncate` call down to the lower layers. A crash may cause that reply to get lost, in which case the deduplication index becomes desynchronized. We solve this by adding an index invalidation bitmap to the DSS. Each set bit invalidates a chunk of the index; a bit is set while any index entry (i.e., object)

in that chunk has a pending downcall. Upon recovery, the cache must reconstruct all entries in the marked chunks. It does so by explicitly requesting the XOR'ed checksum from the lower layers for each affected object, using `getattr` calls.

Third, in order to ensure eventually-perfect deduplication even in the presence of cache crashes, the duplication candidates list must be preserved across restarts. Thus, it would have to be tracked by the DSS, which would be simple but costly. We have not yet added such recovery support.

7.4.5 Discussion

We now turn to the questions posed in the introduction. First of all, we have shown that it is indeed feasible to construct a storage stack with advanced functionality and a strong focus on reliability, by combining several partial solutions. Even though the integration forced us to depart from several simplifying assumptions made in earlier work, the subsequent changes have not compromised any of the original techniques. In addition, this study further strengthens our notion that when considering the bigger picture, individual building blocks such as checksums, recovery points, and transactions can be exploited for multiple reliability purposes.

Our resulting stack has several limitations. For example, we currently assume that software crashes of the individual layers are independent. Therefore, we deliberately choose not to protect the logging data structures with the DSS in the cache, since this would introduce more complexity for the unlikely event of crashes of multiple layers at once. However, since failures may propagate to other layers, the assumption may not hold in practice. Thus, the stack may benefit from more thorough checks at the layer boundaries (along the lines of Recon [41]). As another example, the only software bug protection we provide for the VFS layer is the isolation inherent to virtualization: instead of taking down the entire system, a VFS crash will now take down only its containing domain. VFS maintains large amounts of application state, making it an interesting target for future research.

It is clear that our integration adds more complexity, mainly to retain crash recovery support for the cache layer. Thus, alternative approaches should be considered for that aspect. It would be possible to place the virtualization and deduplication functionality in a new layer above the cache. However, this would not simplify the problem: in order to support recovery from a crash, the new layer would have to be stateless, modifying the transactions it forwards on the fly. That is effectively what our changes to the cache layer do. Thus, aside from the performance overhead of an extra layer in the critical path, such a new layer would serve to make our implementation cleaner, not less complex. Instead, we believe that we could reduce the added complexity by using generic post-crash rollback functionality, for example based on work by Vogt et al [156]. In order to keep runtime overhead low, this technique would require integration with our high-level transaction optimizations; that is part of future work.

7.5 Evaluation

In this section, we provide further answers to our questions by evaluating our work. We first evaluate the performance of vLoris (Sec. 7.5.1). We then continue with rvLoris, measuring its performance (Sec. 7.5.2) and reliability (Sec. 7.5.3).

7.5.1 vLoris performance

We use storage macrobenchmarks to evaluate the performance impact of our virtualization changes to the storage stack. In all performance experiments, we use an Intel Core 2 Duo E8600 PC with 4 GB of RAM and a 500 GB 7,200 RPM Western Digital Caviar Blue SATA test disk, and a version of MINIX 3 that implements our new virtualization support. The benchmarks are run from a single domain. We configure the Loris naming layer to enable access time updates, limit the Loris cache layer to 1 GB of memory, and use PhysFS in the Loris physical layer, operating on the first 32 GB of the disk. The system performs a `sync` once every five seconds and whenever 10 % of the page cache has become dirty.

We use the following benchmarks and configurations: an OpenSSH unpack-and-build test which unpacks, configures, and compiles OpenSSH in a `chroot` environment; a source compilation of MINIX 3 in a `chroot` environment; PostMark, with 800K transactions on 40 K files across 10 directories, using 4 to 28 KB file sizes and 512-byte unbuffered I/O operations; FileBench File Server, single-threaded, run for 30 minutes at once; and, FileBench Web Server, single-threaded, changed to access files according to a Zipf distribution ($\alpha = 0.98$), run for 30 minutes as well. Most of the benchmarks run (almost) entirely in memory, which is the worst case for most of our changes.

We run the benchmarks on several Loris versions along its transition into vLoris, in the order as described in Sec. 7.3. With physical-layer checksumming turned off, we first run the original Loris (*Baseline*), to which we incrementally add object virtualization and copy-on-write support (*Virtualization*), support for naming-layer transactions (*Transactions*), and attribute localization (*Attributes*). On top of these changes, we turn on checksumming with Fletcher's checksum algorithm (*Checksums*), and add deduplication support in three steps: the maintenance of the in-memory deduplication index and the candidates list (*Index*), comparison of candidates (*Compare*), and actual deduplication (*Merge*).

We run each combination of benchmark and Loris configuration at least five times and report the average. The results are shown in Table 7.1. The numbers in parentheses represent performance relative to the baseline. For the OpenSSH and MINIX 3 build benchmarks, lower is better. For PostMark and the two FileBench benchmarks, higher is better.

Since the benchmarks are run from a single domain, the small overhead of object virtualization and copy-on-write support (*Virtualization*) is due entirely to record keeping of the domain mappings and reference counts.

Benchmark	Baseline	Virtualization	Transactions	Attributes	Checksums	Index	Compare	Merge
OpenSSH build	597 (1.00)	600 (1.01)	616 (1.03)	622 (1.04)	617 (1.03)	614 (1.03)	623 (1.04)	629 (1.05)
MINIX 3 build	824 (1.00)	833 (1.01)	842 (1.02)	847 (1.03)	843 (1.02)	842 (1.02)	851 (1.03)	843 (1.02)
PostMark	258 (1.00)	255 (0.99)	245 (0.95)	243 (0.94)	243 (0.94)	243 (0.94)	244 (0.95)	183 (0.71)
File Server	2423 (1.00)	2421 (1.00)	2541 (1.05)	2499 (1.03)	2489 (1.03)	2463 (1.02)	450 (0.19)	324 (0.13)
Web Server	17731 (1.00)	17537 (0.99)	17479 (0.99)	17493 (0.99)	17474 (0.99)	17534 (0.99)	17520 (0.99)	17631 (0.99)

Table 7.1: Macrobenchmark performance of vLoris. The OpenSSH and MINIX 3 build results are in seconds (lower is better). The PostMark results are in transactions per second (higher is better). The File Server and Web Server results are in operations per second (higher is better).

Benchmark	TwinFS	CR-Naming	CR-Cache	CR-Lower	MD-Cache	Index′	rvLoris
OpenSSH build	657 (1.10)	650 (1.09)	657 (1.10)	656 (1.10)	659 (1.10)	656 (1.10)	663 (1.11)
MINIX 3 build	890 (1.08)	887 (1.08)	888 (1.08)	892 (1.08)	913 (1.11)	900 (1.09)	917 (1.11)
PostMark	235 (0.91)	235 (0.91)	229 (0.89)	236 (0.92)	230 (0.89)	235 (0.91)	226 (0.88)
File Server	2332 (0.96)	2335 (0.96)	2299 (0.95)	2322 (0.96)	2299 (0.95)	2295 (0.95)	2269 (0.94)
Web Server	17297 (0.98)	17325 (0.98)	17313 (0.98)	17441 (0.98)	16979 (0.96)	17431 (0.98)	16867 (0.95)

Table 7.2: Macrobenchmark performance of rvLoris.

The addition of the transactions support (*Transactions*) has mixed effects. In particular, PostMark takes a hit while File Server speeds up. Compared to our earlier evaluation [152], our new PostMark configuration performs subpage writes, thereby exposing an implementation-specific overhead of such writes: when wrapped in transactions, these writes require an extra kernel call. We are working on resolving this issue.

The File Server speedup is a result of the change in how access times are updated in the naming layer, which in turn affects object caching behavior in the cache layer. This result is specific to the combination of File Server's access patterns and our cache size, and does not extend to other benchmarks.

The localization of attributes (*Attributes*) has a small effect on performance. In the physical layer, the attributes are now stored as data and no longer part of inodes, which could have a negative effect on locality. However, the cache layer colocates the attributes with their corresponding domain mapping entries, thus achieving another form of locality.

The *Checksums* column shows that enabling Fletcher checksumming–as needed for deduplication–adds practically no overhead, although this can be expected with benchmark configurations that mainly stress the cache. The *Index* column shows that maintaining the index and generating candidates has no further impact on performance, thus confirming that the inline part of our deduplication system has little overhead.

When comparing candidates against potential matches (*Compare*), File Server shows significant overhead. During runtime, PostMark and File Server both generate new files that have all-zeroes contents. Thus, for both benchmarks, all same-sized files end up being flagged as candidates for deduplication. For PostMark, the entire workload fits in the page cache, and thus, the contents comparison operates on cached pages only. For File Server however, the candidate comparison thrashes the cache by reading the matches from disk. Sec. 7.3.4 described several possible improvements.

The last column (*Merge*) shows that the same two benchmarks suffer even more from actual deduplication. Merging objects is cheap, but a large number of merged objects end up being appended to later, forcing the storage stack to copy the underlying object again.

In general, it can be expected that making copies of entire objects is expensive. As we mentioned before, vLoris could be extended with subobject (block-level) copy-on-write and deduplication support. In this work, we have presented the necessary infrastructure to support object-level virtualization in Loris, and if we discount the deduplication comparison and merging, our changes impose between -2 and 6 % performance overhead for the workloads in our tests.

7.5.2 rvLoris performance

For rvLoris, we start with the vLoris *Checksums* version, but we switch from PhysFS to TwinFS, with a 16-block twin offset (*TwinFS*). In terms of reliability, this configuration includes support for disk corruption detection and whole-system failure recovery. On top of the *TwinFS* configuration, we measure the individual performance of the following extensions: crash recovery of the naming layer (*CR-Naming*), the cache layer (*CR-Cache*), and the lower layers (*CR-Lower*); detection of memory corruption in the cache layer (*MD-Cache*); and, again, maintenance of the deduplication index and candidates list (*Index′*). Finally, we enable all extensions at once (*rvLoris*). The results are shown in Table 7.2, including numbers relative to the original *Baseline* from Table 7.1.

When compared to the earlier *Checksums* results, the *TwinFS* results show a 1–7 % overhead. While no worse than our original results (Sec. 7.2.2), this is substantial. Other on-disk consistency formats may be able to reduce the overhead.

Since vLoris already includes transaction support, crash recovery of the naming layer (*CR-Naming*) only adds writing markers to avoid repeating requests after recovery (Sec. 7.2.4). Since these write operations can always be combined with an existing transaction, this feature adds very little overhead.

The new cache crash recovery (*CR-Cache*) initially had high overheads, caused by extra checksumming: every small change to a metadata object page (e.g., changing a reference count) requires immediate full-page rechecksumming for the DSS. We implemented subpage checksum update support for Fletcher, which reduced the overheads by a large margin. The remaining overhead is still due to the cost of Fletcher checksumming in software; in our previous experiments (Sec. 7.2.3), we used checksumming support in hardware.

The cache-layer logging for lower-layer restarts (*CR-Lower*) has little overhead. The sync policy which keeps the system responsive by not allowing the cache to build up too many dirty pages, also prevents that flushed dirty pages kept around for the log put a strain on the cache.

The memory corruption detection (*MD-Cache*) overheads are due entirely to checksumming, like with *CR-Cache*. Deduplication indexing and candidate generation (*Index′*) yields overheads similar to those of the earlier runs (*Index*).

Given that *rvLoris* combines all extensions, it is not surprising that it performs slightly worse than the worst-performing extension. Compared to the *TwinFS* results, rvLoris has an overhead of 1–3 %. Thus, even after our integration efforts, the overheads of the reliability improvements remain low.

Overall, the transition of the original Loris, with no virtualization or reliability features, into rvLoris, which incorporates virtualization, deduplication indexing, and resilience against all four reliability threats, adds an overhead in the 6–12 % range. We believe these numbers are quite reasonable.

7.5.3 rvLoris reliability

Finally, we evaluate the reliability of rvLoris. Due to space constraints, we report on the area affected most by this work: resilience against software bugs. As a side effect, we test recovery points and thus whole-system failure recovery. Additional experiments have confirmed that our protection against memory and disk corruption has not been affected by the new changes. We do not include those results here.

We perform fault injection experiments on each of the four rvLoris layers, while running either an OpenSSH build or a version of PostMark which verifies all file data and call results. Using the framework from previous work [152], we inject two types of faults: fail-stop faults which merely crash the process (1,000 injections per configuration), and random faults which simulate the effects of common software bugs (250 injections per configuration). Once per minute, we inject 100 faults at once. Thus, in total, we inject one million faults.

We measure the number of resulting crashes in the target layer, successful recoveries, permanent failures, timeouts, crashes in other layers, and failures propagated to applications. In theory, the *fail-stop* faults should always lead to successful recovery, except in the cache layer, which may flag permanent failure due to a checksum error in the DSS; after all, DSS updates are not atomic (Sec. 7.2.3). The *random* faults may however cause silent failures, thereby violating assumptions we made in all our works; in particular, that corrupted results are never propagated across layers. Because of the resulting risks, we perform these experiments in a virtual machine.

The results are shown in Table 7.3. For *fail-stop* fault injection, all injections in the naming, logical, and physical layers indeed resulted in a crash and then successful, application-transparent recovery. As expected, in a small number of cases, cache-layer recovery failed on a DSS checksum mismatch. In addition, some of the cache-layer crashes caused a cross-layer memory copy failure, resulting in a crash of the logical layer due to inadequate error handling. In all other cases (98 %), the cache layer recovered successfully.

As expected, the *random* fault injections resulted in more diverse failures, including cases where no effects were observed at all. In several cases, the faults caused operations to start returning errors, thus resulting in propagation of corrupt results to other layers and often also to the application. Again, such failures are beyond the scope of our work. In many other cases, the faults caused a request or reply to be dropped, resulting in no progress (timeout); call timeouts could fix this. In the majority of cases (86 %) however, our crash recovery techniques caused the random fault injection to result in application-transparent recovery.

7.6 Related Work

Our previous papers already provide overviews of work related to their respective topics. Here we discuss work related to combinations of reliability in the storage stack.

Layer	Benchmark	Fail-stop fault injection							Random fault injection						
		I	C	R	P	T	L	A	I	C	R	P	T	L	A
Naming	OpenSSH	1000	1000	**1000**	0	0	0	0	250	223	**216**	0	9	1	6
	PostMark	1000	1000	**1000**	0	0	0	0	250	228	**226**	0	22	0	2
Cache	OpenSSH	1000	1000	**989**	9	0	2	0	250	222	**201**	4	28	1	16
	PostMark	1000	1000	**980**	13	0	7	0	250	226	**214**	6	24	0	6
Logical	OpenSSH	1000	1000	**1000**	0	0	0	0	250	212	**212**	0	38	0	0
	PostMark	1000	1000	**1000**	0	0	0	0	250	219	**210**	0	31	1	8
Physical	OpenSSH	1000	1000	**1000**	0	0	0	0	250	222	**221**	0	28	1	0
	PostMark	1000	1000	**1000**	0	0	0	0	250	230	**228**	0	19	0	2

Table 7.3: Fault injection results, showing per layer, benchmark, and fault injection type: the number of times fault injection was performed (I), and the resulting number of target layer crashes (C), successful recoveries (R), permanent failures (P), timeouts (T), other-layer crashes (L), and application failures (A).

Membrane [138] implements support for checkpointing and logging for Linux file systems. Membrane can make use of file system support for recovery points, although this requires small changes to such file systems for fully correct recovery.

EnvyFS [17] uses N-version programming to protect applications from certain classes of software bugs and disk corruption, by employing multiple different file systems to perform the same actions. EnvyFS has substantial performance overheads and complicates system crash recovery.

Z^2FS [168] is a variant of ZFS that can switch between checksum types to detect both memory and disk corruption at low cost, although also with lower detection guarantees.

Various generic techniques based on in-memory transaction [93] or checkpoints [156] offer the potential to recover from software bugs across the entire storage stack at once, with guarantees similar to those we provide for the cache. However, given that such techniques inherently make rollback copies for every page written to in the cache, they have exorbitant overheads when applied to the storage stack [93]. As stated in Sec. 7.4.5, we believe a hybrid approach could help here.

7.7 Conclusion

In this paper, we have attempted to provide a first set of answers to questions regarding the integration of several reliability techniques and other functionality in the storage stack. In the process, we have added support for virtualization to our stack. The case study has yielded mostly positive answers, as well as new areas warranting more research.

We believe that several of our findings are applicable beyond this case study. For example, our storage architecture and reliability improvements could be implemented in a monolithic environment–the latter by building on existing recovery techniques (e.g., [140]). More generally, any storage stack faces similar reliability threats, and we expect that our findings regarding the feasibility, advantages, limitations, and complexity of adding comprehensive reliability support largely apply to other storage stacks as well.

8 Discussion

In this chapter, we address a number of technical points that could not be included in the original papers, mainly due to space constraints. In Sec. 8.1, we discuss a number of technical limitations of our storage device failure detection and recovery as originally covered in Chapter 2. In Sec. 8.2 and Sec. 8.3, we discuss two related limitations of our whole-system recovery system from Chapter 3: checkpoints and the freeze window, and the `fsync` problem, respectively. In Sec. 8.4, we describe a possible improvement to the process crash recovery system of the cache layer described in Chapter 4. Finally, we provide some statistics on the implementation complexity of all our projects in Sec. 8.5.

8.1 On storage device failures

In Chapter 2, for storage device failures, we have focused on effective, low-cost failure detection. As a result, we have adopted the use of *parental checksumming*. We first briefly summarize the reason for the low cost of parental checksumming, and then elaborate on its effectiveness.

Due to the delayed checksum computation described in Sec. 2.4.1, the parental checksumming scheme causes very few extra disk writes. Whenever a file is written to, its file modification and change times are updated as well, thus causing an update of the attributes in the file's inode record. As a result, updating checksums in the same inode record does not infer extra disk writes. Practically speaking, the only extra write activity of the parental checksumming comes from overwrites in existing blocks referred to from indirect blocks. The "layout only" results in Sec. 2.6.3 confirm that the practical overhead of writing checksums to disk is negligible.

The parental checksumming approach guarantees detection of disk corruption, although not necessarily immediately: when a data block is corrupted while being written to the storage device, the corruption may be detected only when the block is

read back from disk, which is potentially much later in time. It would be possible to read back each block immediately after writing it, but this would be prohibitively expensive. However, the parental checksumming scheme does guarantee that the corruption is detected before the block is ever propagated to higher layers, thus preventing that "fail-partial" disk drive behavior is exposed to the logical layer. In Chapter 2, we refer to the physical layers' model of returning either a corruption-free block or an error in response to a read request as "fail-stop."

Thus, parental checksumming detects all forms of disk corruption, but with one notable exception: the case that the storage device is consistently failing silently to write data to the device at all. In that case, the resulting disk contents are not corrupt (all checksums match), but stale. This could lead to problems when the contents of multiple devices are expected to be synchronized. Our original PhysFS implementation takes a number of steps to counter this problem. When writing the super and root blocks to the device, our Loris implementation instructs the disk driver to bypass its write-back cache, but such bypass instructions are not always honored by the disk drive [119]. In addition, Loris can be configured to perform a read-back test of these blocks after writing, but it is conceivable that the disk serves such requests from its write-back cache, without making any changes to the device. Without involving multiple devices in the solution, this case can never be fully prevented. With multiple devices, the problem is fully resolved as a side effect of the checkpoint timestamp scheme introduced in Chapter 3. Therefore, we believe that the exception does not pose a serious threat in practice.

More generally, failure detection implicitly assumes that the checksums themselves are strong enough to detect data modification. There is a small possibility that an accidental modification still results in a valid checksum. In that case, the use of a stronger (but typically slower) checksum algorithm would suffice for improving detection. A modification may also be malicious, in that the storage device driver, firmware, or hardware is actively rewriting checksums. Such cases fall outside our assumptions and require a different approach, for example based on cryptographic hashing or even encryption of storage contents.

For recovery purposes, we rely on the presence of redundancy across multiple devices. However, many systems can be expected to have only a single storage device. We have not discussed this case, because it does not allow for recovery in the case of a whole-device failure. However, it would be possible to improve on the current situation with respect to partial device failures, for example by introducing forms of intradevice redundancy [90]. Such a feature could be implemented within a physical module, without involvement of the rest of the stack. A single-device system could also invoke a `fsck`-like checker as soon as a problem is detected. It could even ignore checksum failures for at least data blocks, if availability is deemed more important than integrity.

8.2 Checkpoints and the freeze window

Our approach for establishing new checkpoints presented in Chapter 3 has a major drawback. If the whole system experiences a failure right before the last module in the physical layer finalizes its (local) on-device checkpoint, then all physical modules must be able to roll back to the (global) checkpoint that they all have in common. In this scenario, for all but one of the physical modules, the common checkpoint is the penultimate on-device checkpoint. As a consequence, a physical module with an on-device layout that supports no more than two on-device checkpoints, such as TwinFS, will have to hold off invalidating the penultimate checkpoint until it knows that all other physical modules have committed the new checkpoint to their respective devices. Thus, for as long as a new global checkpoint is being established, TwinFS can effectively make no other changes to the device. This may cause a loss in performance. This *freeze window* is as long as the difference between the shortest and longest time of taking a checkpoint across all the physical modules; to be precise, the shortest time is determined by the physical modules with support for no more than two checkpoints. This shortcoming can be alleviated by using only physical modules that support three or more on-device checkpoints.

In our prototype, the **sync** operation acts like a barrier, blocking any other operations across the entire Loris storage stack while it is being processed. While that puts it behaviorally on par with the MFS file system implementation used as baseline in Chapter 2, the resulting performance loss may extend well beyond the theoretical effects of the freeze window, for several reasons. First, our implementation's **sync** operation blocks not only modifying operations, but also read-only operations. Second, modifying operations are not propagated to the physical layer until the global finalization of the checkpoint, while a physical module might already be able to aggregate operations and for example perform necessary metadata read-ahead if it gets the modifying operations immediately. Third, the implementation needlessly applies the freeze to physical modules which do support more than two checkpoints on their respective device.

It would be possible, but not trivial, to remove this limitation from our prototype. In addition to making the layers' internal implementation more complex, it would require a (small) extension to the protocol used between the logical and physical layers. With the current approach, the global finalization of each checkpoint is implied by the moment that a physical module gets a new operation after replying to the **sync** operation. If a physical module could get new operations after the local, but before the global checkpoint finalization, then the logical-physical protocol would have to be extended with a notification to indicate global finalization, implying that the penultimate checkpoint may now be discarded.

Also, we stress that TwinFS has been a proof-of-concept implementation only, designed primarily for simplicity. We believe that more advanced checkpointing schemes, for example based on journaling or logging, could not only eliminate the freeze window, but also offer significantly better performance than TwinFS.

8.3 The fsync problem

The existence of one particular POSIX system call, `fsync` [109], poses an additional problem for the checkpointing solution of Chapter 3–we call this the *fsync problem*. The `fsync` call is supposed to flush all changes involving the open file indicated in the call. The Loris checkpointing infrastructure makes it difficult to perform fine-grained flushing, and our Loris prototype therefore translates each `fsync` call into a full `sync` call, flushing all changes of all files instead. With many `fsync` calls coming from applications, such coarse-granular flushes can become rather expensive. Therefore, it is worth analyzing exactly why `fsync` presents a problem for Loris, and whether there are potential solutions.

The main problem with selective cache flushing in Loris is that the stack is not designed to retain a level of independence between operations that is sufficient to prevent inconsistencies. For example, in order to flush the contents of a new file, its new size has to be flushed as well. This means that upon getting an `fsync` call, the target object's attributes have to be flushed. However, these attributes include both the file size and the file link count. By design, the meaning of the attributes is known only in the naming layer, whereas multiple `setattr` operations on the same object are aggregated and merged in the cache layer for efficiency reasons. Now, suppose an application first creates a new hard link for a file (thus increasing its link count), and then calls `fsync` on that file. The naming layer would relay the `fsync` operation to the cache layer. In the cache layer, flushing just that object's data and attributes is clearly not enough: this would flush the new link count, but not the corresponding entry in the directory, thus introducing inconsistency.

The inconsistency problem could be solved by extending the flush operation to all objects that were ever part of a transaction that included the target object, but the transitive set is likely to include most if not all object changes, thus making the `fsync` call still as expensive as a full `sync` call. While this particular problem can be solved in the cache layer with extra information in the transactions, that would merely move the problem to the physical layer, which typically aggregates state for multiple objects together in a single disk block (e.g., a bitmap or inode block), thus also creating a similar problem. Only full and fine-grained transaction support, including transaction history information for dependency tracking, extended all the way into the physical layer, could solve the entire problem. Such a change would come at great expense, in both performance and complexity.

An alternative and much simpler approach would be for the cache layer to write its entire operations log, as added for crash recovery in Chapter 3, to the actual storage devices. This approach would not limit the amount of pending data to be flushed down, but it may speed up the flushing itself, especially if the target of the write is a special metadata object that uses a preallocated, contiguous area on the lower devices. After all, contiguous disk writes are relatively inexpensive. The approach resembles journaling, and would require only two extensions: 1) the physical modules have to support the contiguous preallocation, and 2) the cache layer has to

support serialization and deserialization of its operations log.

8.4 Improving cache-layer recovery

For the purpose of crash recovery, the cache-layer recovery system of Chapter 4 makes a number of assumptions about the misbehavior resulting from software bugs. In particular, even though the recovery system discards most state changes made since the start of the ongoing request, it still makes the assumption that during the processing of that request, no corrupted state has propagated into either the lower layers or the Dirty State Store (DSS). After all, operations sent down to lower layers cannot always be undone, and while the DSS checksums protect against unintended memory overwrites in the DSS memory area, they do not protect against legitimate but corrupt calls into the DSS part of the cache layer.

The issue could be mitigated by deferring all the calls to lower layers and modifications to the DSS until the end of each request. This approach maximizes the chance that in the event of a crash, no propagation has taken place yet, and thus, no corrupted state could have been propagated either. The approach is very similar to generation of transactions in the naming layer. However, in the case of the cache layer, deferring actions may be problematic.

The most problematic case involves writing to cache pages. An update of a dirty page must be paired with an update of its DSS checksum; after all, if the two do not match, the cache will not be recovered after a crash. Thus, in order to defer updating the DSS checksum until the end of a request, the update of the dirty page must be deferred correspondingly. This means that cache pages can no longer be updated in-place as part of processing `write` operations. The cache layer would need temporary buffers to store updates to pages, and extra memory copy operations to merge the updates into the actual pages at the end of the requests (or a similar approach with the same costs). This would negatively affect both performance and complexity in the cache layer.

However, there are some mitigating factors. First, temporary buffers are needed only for non-append writes to pages that were already dirty. The idea here is that if the DSS does not already have a checksum for the (part of the) page being changed, no mismatch will exist after a crash either. If need be, a checksum entry can be thrown out of the DSS immediately if it is determined that the new write call will overwrite the entire previously checksummed area, thus also avoiding the need for temporary buffers in the case of full-page overwrites of already-dirty pages. The post-crash repeat of the active write calls will do the rest. Second, it would be possible to compute new page checksums while copying memory between buffers.

Time constraints have prevented us from experimenting with this alternative approach.

8.5 Implementation complexity

Due to space constraints, we have not been able to report statistics about the source code changes in our original papers. However, we believe that especially for reliability, it is worth elaborating on exactly this aspect, since every source code addition may itself contain new bugs. In this section, we report on changes in source code, using source code numbers obtained with the SLOCCount tool [162]. We report source lines of code (SLOC) counts on a per-module basis; the continuous internal reorganization of the Loris source code makes it less useful to report finer-grained statistics. In addition to the modules, we also report statistics for the Loris library, which is shared between all these modules. The library's main task is to handle all aspects of intermodule communication, so that modules can either be placed into separate processes or (as shown in Chapter 2 only) be merged into a single process, without changing the modules themselves. The library also contains a certain amount of other shared code, such as checksumming algorithms.

For each of the projects in Chapters 3 to 7, we report the SLOC counts of Loris before and after a particular project, and the delta which is simply the difference between the two. Changes in the *Before* numbers across tables are not meaningful. Loris has served as a base for several other projects [7], and many changes have been merged into the main Loris code at arbitrary times. In other cases, we have later removed features which became too cumbersome to maintain; an example is background flushing in the cache layer. As a result, the baseline Loris code is minimalistic in many ways, and often much simpler than would be possible for a production-grade implementation. Also, it should be noted that in several of our projects, the same module implementation supports all variants of the algorithms presented in the papers, using systems of `#define`s. For these reasons, the code size changes should be considered an upper bound in most cases.

Table 8.1 shows the SLOC counts for the system and process crash recovery work in Chapter 3. In contrast to the other tables in this section, Table 8.1 contains an approximation: at the time, the work of Chapter 3 was merged with the main Loris source code immediately, in small pieces that are intermixed with a large number of other changes, including support for multithreading, support for versioning [148], and a redesign of the block device driver interface. Except for the TwinFS

Component	Before	After	Delta
Naming	2,859	2,864	+5
Cache	1,818	2,694	+876
Logical	3,143	3,338	+195
PhysFS	3,071	3,079	+8
TwinFS	0	3,524	+3,524
Library	8,662	8,867	+205

Table 8.1: SLOC counts of the changes made for Chapter 3.

Component	Before	After	Delta
Naming	3,165	3,211	+46
Cache	2,564	3,284	+720
Logical	6,332	6,339	+7
PhysFS	2,659	2,697	+38
TwinFS	3,438	3,459	+21
Library	8,842	9,195	+353

Table 8.2: SLOC counts of the changes made for Chapter 4.

numbers, the *Before* numbers are from the Loris source code just before making the first crash recovery changes, and the *After* numbers are the sum of deltas for two branches (hand-created for this purpose) which contain just the changes related to crash recovery, logging, and resynchronization, added to the *Before* numbers. We consider this the most accurate approximation for the SLOC counts before and after the project of Chapter 3 that we can make. TwinFS was created specifically for this chapter, and thus, its SLOC count is accurate.

The SLOC increase in the cache layer is due mostly to the cache logging implementation. The cache logging code is rather simple, but it maintains its own data structures, needs to support each of the modifying Loris operations, and comes with support for replay, and thus still ends up representing a substantial increase in overall code size. The logical layer was extended with support for coordinating checkpoint loading and data resynchronization, and the Loris library was extended with support for new operations to support this coordination. TwinFS was designed as a straightforward extension of PhysFS, and is therefore able to reuse most of the PhysFS code with no change. The difference in code size between PhysFS and TwinFS is largely due to two aspects: 1) support for the resynchronization log, and 2) support for built-in self-verification of operational correctness, by checking at runtime that no stable blocks are overwritten. The self-verification requires additional CPU and memory resources, and was therefore disabled during all our experiments.

Table 8.2 contains a (fully accurate) representation of the changes made for Chapter 4. The main change to the library is the addition of support for checksum

Component	Before	After	Delta
Naming	3,012	3,561	+549
NamingHD	0	2,468	+2,468
Cache	2,564	2,861	+297
Logical	6,332	6,345	+13
PhysFS	2,658	2,666	+8
TwinFS	3,438	3,446	+8
Library	8,842	9,491	+649

Table 8.3: SLOC counts of the changes made for Chapter 5.

Component	Before	After	Delta
Naming	3,073	3,130	+57
Cache	2,485	3,316	+831
Logical	6,359	6,396	+37
PhysFS	2,674	2,688	+14
TwinFS	3,553	3,563	+10
Library	8,934	9,038	+104

Table 8.4: SLOC counts of the changes made for Chapter 6.

propagation through the stack. The cache-layer code increase is due to two main changes: 1) support for checksum generation and verification throughout the code, and 2) the implementation of the Dirty State Store, including not only management but also post-crash verification of its checksummed data structures.

The code changes for Chapter 5 are shown in Table 8.3. The primary Loris namespace module, Naming, has been extended with support for optimistic transaction generation and corresponding support for rollback of internally cached data, as well as support for request recovery records. NamingHD is our HDF5 extension namespace module. The number reported for this module is the custom written glue code only. It does not include the HDF5 library itself, of which we used version 1.8.10-patch1; it has a SLOC count of 455,157. The difference in the Loris library is the result of (de)serialization support for transactions. Similarly, the cache-layer changes consist of support for atomically processing transactions.

In Chapter 6, we introduced basic object copy-on-write support. This support is implemented in the cache layer, and Table 8.4 shows a corresponding increase in the cache-layer code size, mainly to maintain the domain mapping objects and the global reference count object. A substantial part of the code changes are due to support for creating and destroying new domain mappings; these changes also affect the naming layer and Loris library. In contrast, the implementation of the `copy` operation largely reused existing code for object versioning, and thus added little code in the physical layer.

Chapter 7 also added basic object deduplication, and Table 8.5 shows the code difference of just this aspect. The main change is in the cache, adding support for

Component	Before	After	Delta
Naming	3,793	3,793	+0
Cache	3,759	4,522	+763
Logical	6,364	6,366	+2
PhysFS	2,661	2,751	+90
TwinFS	3,536	3,624	+88
Library	9,963	9,956	-7

Table 8.5: SLOC counts of the deduplication changes made for Chapter 7.

Component	Before	After	Delta
Naming	3,073	3,793	+720
Cache	2,485	4,382	+1,897
Logical	6,359	6,366	+7
PhysFS	2,674	2,751	+77
TwinFS	3,553	3,624	+71
Library	8,934	9,956	+1,022

Table 8.6: SLOC counts going from Loris to vLoris in Chapter 7.

hashing and indexing, maintaining a candidates list, equivalence testing, and object merging. The physical modules were extended with support for returning block checksum differences resulting from `write` and `truncate` operations. We note that in this case, the *Before* numbers already include a large number of reliability changes, on top of which we added the deduplication support (but without its corresponding integration with the reliability changes).

Table 8.6 shows the statistics for the transformation of Loris into vLoris, as described in Chapter 7. The *Before* column shows the SLOC counts for the original Loris which, as explained, already contains the changes from Chapter 3. The *After* column shows the SLOC counts after adding support for object copy-and-write and deduplication, transactions, lazy access time updates, and attribute localization. For our own convenience, we already integrated all naming-layer restart support from Chapter 5 along with the support for transactions into vLoris, even though the support (in particular, the generation of request recovery records) was disabled during our vLoris experiments. This, in addition to the lazy access time support, is the reason for the code increase in the naming layer. As expected, the cache-layer SLOC count increase is close to the sum of the deltas in Tables 8.3, 8.4, and 8.5. The same holds in part for the Loris library, but we also added support for Slicing-by-8 CRC32 checksumming [1] to the library while constructing vLoris.

Finally, Table 8.7 shows the statistics for the transformation of vLoris into rvLoris. In this case, the *Before* column shows the SLOC counts from vLoris (Table 8.6), and the *After* column shows the SLOC counts for rvLoris. In terms of source code, the transformation consists of two parts: 1) the integration of the checksum propagation

Component	Before	After	Delta
Naming	3,793	3,841	+48
Cache	4,382	6,163	+1,781
Logical	6,366	6,368	+2
PhysFS	2,751	2,794	+43
TwinFS	3,624	3,694	+70
Library	9,956	10,401	+445

Table 8.7: SLOC counts going from vLoris to rvLoris in Chapter 7.

and cache-layer checksumming code from Chapter 4, for which the SLOC counts are shown in Table 8.2, and 2) all the integration changes as described in Chapter 7. Most notably, the difference in the deltas shown in Table 8.2 and 8.7 show that the integration effort required roughly a thousand extra lines of source code in the cache layer.

9 General Conclusions

In this dissertation, we have investigated the construction of a highly reliable storage stack. We have first presented a rearrangement of the traditional storage stack. The new arrangement, called Loris, has advantages in the areas of reliability, heterogeneity, and flexibility. For reliability in particular, it offers detection of all the main forms of silent storage device failures. In the lower layers of the Loris stack, we have developed a solution for whole-system failures and software bugs, both based on a system for creating and loading checkpoints, with a new on-disk layout to manage such checkpoints as proof of concept. We have shown how the cache layer of the Loris stack can both detect memory corruption and recover from the effects of software bugs with a system that reuses checksums already present in the storage stack, thereby limiting the overhead of the solution. We have shown how the naming layer of the Loris stack can offer extended functionality, and presented an approach that allows both the original naming module and such new extensions to recover from crashes, using transactions to make these naming modules effectively stateless at low cost. We have shown how the effects of a crash in the storage stack's Virtual File System (VFS) layer, as well as other components in the operating system, can be limited through isolated replication, and we have extended this concept into a new approach to virtualization, with a number of advantages compared to the established virtualization alternatives. Finally, we have integrated all pieces to form a reliable, virtualizing version of the Loris storage stack, providing better reliability against storage device failures, whole-system failures, memory corruption, and software bugs, at a cost of performance, resource usage, and complexity that we believe is within reason for computer systems for which improved reliability is desired.

We have implemented our techniques on the MINIX 3 platform, which we have shown to be highly suited for our purposes. This choice has allowed us to investigate the limits of our reliability improvements without being held back by

implementation-level difficulties. However, we believe that while this particular platform was the right choice for our work, many aspects of our work are also applicable to other systems. First of all, given the many advantages of the Loris storage stack arrangement over the traditional storage stack, it is our point of view that every operating system with the traditional storage stack could benefit from a similar rearrangement, even though we realize that this may not be a trivial task in many cases.

Similarly, most of the reliability improvements could be applied directly in any implementation of the Loris arrangement. There is one group of exceptions: the solutions for recovery from process crashes, which rely heavily on the isolation provided by our base platform. However, other research has shown that relatively low-cost forms of isolation can be achieved in monolithic systems [138, 140]. Such isolation is sufficient to allow implementation of our recovery techniques, although the provided level of isolation is not as strong and therefore not able to provide the same protection and detection guarantees, for example with respect to execution of unprivileged instructions, wild writes, and infinite loops. Thus, while all our reliability techniques could in theory be implemented on a monolithic system, we believe that the microkernel architecture is the right design for the operating system of any system requiring high reliability or security, and thus worth considering even if it comes at the expense of degraded performance.

In contrast, our virtualization idea does not lend itself well to being implemented on the majority of the currently existing operating systems. That is intentional: we have argued that by letting go of the more traditional system organization, we can achieve virtualization properties not found in other systems. With that said, Chapter 6 has shown that some virtualization platforms have already taken small steps in the same direction, by breaking down some of the traditional information barriers between the operating system and the hypervisor–a trend that, we hope, will continue. Regardless of whether it will come from radical redesign or through incremental steps, we believe that there will be a place for our idea in the future landscape of virtualization technologies.

In the rest of this chapter, we provide answers to our original research questions (Sec. 9.1) and discuss possible avenues for future research (Sec. 9.2).

9.1 Answers to research questions

We now return to the research questions posed in the general introduction. Since these research questions touch upon issues that are highly cross-cutting, they could not be answered fully in the individual chapters, with one exception: the first question, regarding the feasibility and properties of an integrated reliability solution for the Loris storage stack, has been answered in Chapter 7. In this section, we answer the remaining questions, regarding building blocks for reliability (Sec. 9.1.1), exploiting high-level knowledge (Sec. 9.1.2), the ideal level of componentization

(Sec. 9.1.3), and robustness against software bugs (Sec. 9.1.4).

9.1.1 Building blocks for reliability

In this thesis, we have shown that several of the solutions to different reliability problems end up using the same basic building blocks to achieve their goal. We reiterate the most important building blocks.

Most importantly, **checksums** have proven to be a particularly versatile component in the storage stack. We have been able to utilize this single building block for four purposes: detection of corruption on storage devices, detection of in-memory corruption in the cache layer, internal state verification after a crash of the cache layer, and candidate generation for object deduplication. While computing checksums is not free, one can choose between various checksum algorithms which occupy different points along the tradeoff spectrum between strength and performance. In this work we have experimented with Fletcher and CRC32–two algorithms with different Hamming distance strengths and performance characteristics. In addition, we have made use of a number of optimizations to limit the overhead of checksum computation, such as tracking partial checksums to optimize for file appends (Chapter 4) and partial checksum recomputation to optimize for small writes (Chapter 7). It has been easy to verify implementation correctness by recomputing and comparing the checksums in various locations throughout the stack.

The system of **consistency points** (checkpoints) serves an equally important role for both recovery after whole-system failures and recovery of crashed processes in the lower layers of the Loris storage stack. As shown in Chapter 3, the design of the recovery system relies on a small number of simple assumptions and protocol additions, leaving the implementation of checkpoints to the individual physical modules. Time constraints prevented us from implementing additional conforming physical modules besides our TwinFS prototype. While related work has shown that on-device support for checkpoints is not strictly necessary for recovery from software failures [138], we have argued in Chapter 3 that such support should be an integral part of the storage stack design. On-device layout schemes that do not support recovery from whole-system failures can safely be categorized as legacy schemes, and as we stated in the general introduction, we deliberately let go of legacy support as a goal. The result is a simpler and cleaner integrated checkpointing system.

The **transactions** system that is implemented in the cache layer and used by the naming layer, is used for multiple purposes as well. First, as shown in Chapter 5, it allows the processes in the naming layer to be restarted after a process crash. Second, as shown in Chapter 7, it allows virtualized domains to recover cleanly from whole-system failures without being actively involved in establishing checkpoints, thus leaving the latter entirely to the host system. As a side effect, the transaction system also guarantees that after a domain failure (e.g., due to a crash of the domain's VFS process), the domain is guaranteed to find a consistent storage state upon its restart. We note that in the context of virtualization, other researchers have independently

reached the same conclusion regarding transactions [72], after we published our idea [149]. We will elaborate on additional functionality suggested in their work in Section 9.2.2.

We believe that the emergence of these building blocks is an important outcome of this work, as it shows that when addressing multiple reliability problems at once, the overall performance loss and complexity incurred by the solutions does not necessarily increase linearly with the number of addressed problems. Therefore, it has indeed shown to be beneficial to consider multiple reliability threats at once.

9.1.2 Exploiting high-level knowledge

Our work has focused on the storage stack instead of the entire operating system, because the narrow focus has allowed us to use high-level knowledge about the storage stack to develop solutions with limited overhead. In this section, we summarize the concrete benefits, as well as the risks, obtained from using high-level, semantic knowledge about the structure, internal operation, and communication primitives of the storage stack for the purpose of improving reliability. We consider differences in terms of strength, performance, resource usage, and complexity, compared to what can hypothetically be achieved by (ideal) generic solutions.

First of all, and as a general point, we rely on the one-directional structure of the storage stack. The storage stack can be seen as the channel between applications and storage devices, and storage devices fundamentally support only idempotent operations. As a result, we have been able to establish idempotence for operations within the storage stack as well, which we have exploited for recovery purposes in Chapters 3 and 4. The linear structure is strengthened by the storage stack's isolation from most other subsystems of the operating system. This allows us to disregard the possibility of side effects which would require special attention during recovery. In particular, any nonstorage resources used in the storage stack are process-bound, meaning that upon restarting a crashed process, they will be cleaned up automatically by the rest of the operating system. As an example, one effect is that our solutions need not track or limit memory usage for storage stack components. We note that while future extensions, such as a physical module for cloud storage, may introduce side effects, the recovery systems will continue to work as long as the idempotence is maintained in such extensions.

For the lower-layer process recovery system of Chapter 3, the cache process manages the data pages in its operations log in a copy-on-write fashion, making an exclusive copy of the page for the log only when the main cache pool is evicting the page. This was trivial to implement, since we could easily identify the code path taken when a page is being evicted, and make the appropriate call into the logging module. In addition, the approach is optimal in terms of performance. In theory, it would be possible for a generic rollback-based process crash recovery approach to implement a similar log, using interception of operations sent down to the lower layers. It would then have to instrument subsequent writes to the pages sent down in

such operations in order to effectively implement copy-on-write semantics for them [156]. It is doubtful that such a generic solution could be implemented with the same minimal overhead as our targeted approach.

Moreover, in Chapter 7, we improve the logging system by making it steal pages from the main cache instead, thus fundamentally altering the operation of the cache for recovery purposes–of course, in such a way that under normal circumstances, the actual performance impact is minimal. In addition, the management of the cache-layer log uses high-level knowledge to reduce the logging overhead even further. For example, it merges writes for the same page of the same object, and discards the log entries of various other operations that have become obsolete by subsequent operations. These optimizations reduce not only resource usage but also implementation complexity. A generic recovery system would lack the necessary semantic knowledge for these changes, and thus, all these improvements can be made with high-level knowledge only.

However, as shown in Chapter 7, we could not retain the most stripped-down version of the log, as the introduction of the `copy` operation brought along the requirement for dependency tracking. Other new operations such as the `snapshot` operation introduced in other work on Loris [11] would require the same kind of dependency tracking. Thus, this is a case where "too much" optimization based on high-level knowledge may clash with new functionality. Still, the added dependency tracking retains the low resource usage of the original solution in the common case, and thus still benefits from using high-level knowledge.

For the cache-layer memory corruption detection and process recovery systems of Chapter 4, a generic approach would involve separate computation of checksums for the integrity checks. In contrast, our solution's reuse of checksums for data on disk obviates the need to recompute these checksums separately. Our system also exploits the fact that clean pages can be reread from disk if corrupted, thus reducing the amount of state for which corruption is fatal to a large extent. Thus, here too, use of high-level knowledge both improves the strength and reduces the overhead of our systems.

For the naming-layer process recovery system of Chapter 5, several high-level optimizations allow the cache layer's transaction processing overhead to be kept to a minimum. One important example is that the transaction system allows each transaction to contain only one operation that cannot be undone (e.g., a `delete` operation), thus reducing overall complexity throughout the storage stack by obviating the need to implement a way to undo such an operation (e.g., with a new `undelete` operation) in the lower layers. Another example is that the cache layer can avoid generating rollback information for `write` operations to data objects (i.e., regular files), which means that no extra memory copies have to be made for these highly common operations.

Summarizing, we have relied heavily on high-level knowledge throughout this work. This knowledge has been essential to some parts of the detection and recovery systems, and contributed significantly in keeping the overheads of these systems low

in other parts. By definition, generic detection and recovery systems cannot use the same high-level knowledge and thus always be a step behind solutions tailored specifically to their targets. As we noted in Chapters 4 and 7, we do believe that extra integration effort between generic and custom systems may be warranted in certain cases, and we consider this to be an important area of future work.

9.1.3 The ideal level of componentization

For the problem of software bugs, and to a lesser extent that of memory corruption, the overall system's modularity and isolation are crucial for the resulting solutions. In contrast to the traditional file system and software RAID layers, the heart of Loris consists of no fewer than four layers. This finer-grained modularity has allowed us to turn the software RAID functionality into an object-based equivalent. We have enumerated the advantages of this change in Chapter 2. Furthermore, the chosen split-up of the layers has resulted in each layer performing a single and clearly defined task. Further splitting up any of the layers would be problematic as multiple layers would end up managing the same data structures, with as result that the simple cross-layer object interface would no longer suffice. Thus, the split-up of the layers has been pushed to the point that further division would yield no advantages. In this section, we look at two subsequent aspects: first, the order of the layers, and second, the relation between layers and processes.

Given the argumentation in Chapter 2, the only aspect of the order of layers that is up for discussion at all, is the position of the cache layer. Even though Chapter 2 states that the Loris cache layer is optional on systems with a unified page cache, such a page cache still needs to be assigned a logical place in the storage stack architecture, even if it is integrated with the VM subsystem. In the Loris stack organization, the overall system benefits from having the page cache below the naming layer, because the natural result is that the page cache also caches directory data. Moreover, our virtualization approach relies on that order as well. On the other side, the page cache befits from being positioned above the logical layer, not only to avoid separately caching redundant copies of data (e.g., for mirrored files), but also to be able to provide unified support for memory-mapped files (which requires logical file information). The main downside of this order is that the physical layer cannot use the page cache for its metadata, but our experiments have shown that this problem is solved adequately with the physical modules each employing their own small metadata cache. In theory, the page cache could be made accessible to the physical layer as well, but this would not change the cache layer's logical position in the stack.

Thus, for our reliability work, we have no reason not to take the original Loris stack architecture and its established modularity as a given. Since the crash recovery systems rely on isolation of processes for the delineation of failure domains, we now turn to the isolation aspect: for reliability purposes, what is the most beneficial split-up of the Loris stack into separate processes? We start with the cache layer.

We have shown throughout this dissertation that the cache layer serves as what

can be called a "stable point" for process crash recovery of the other layers, due to three characteristics. First, in its final form, the cache layer implements transaction support and the operations log which are needed for recovery of the naming layer and the lower layers, respectively. Second, the cache layer is fundamentally the least complicated layer in the stack. Since it provides caching functionality only, its output operations (sent to lower layers) are very similar to its input operations (received from upper layers), and its core data structures and algorithms are simple. Third, its process crash recovery guarantees are the weakest of the four aforementioned layers, due to the fact that it typically contains a large amount of state of which there is no other copy in the stack.

Any functionality that is colocated in the cache layer will inherit this layer's relatively weak process crash recovery guarantees. Any extra functionality will also introduce more code that could have software bugs, thus making the cache process more likely to fail–possibly in an unrecoverable way. These are strong arguments in favor of keeping the other layers outside of the cache-layer process, which would imply at least one process above and one process below the cache layer. We believe that Chapters 2 and 5 make sufficiently clear that both these sides (the naming layer side, and the logical/physical layer side) are indeed sufficiently complex and thus prone to failures to warrant being split off into separate processes. As shown in Chapter 6, the hard split between the naming and cache layers is even fundamental to our virtualization ideas.

Thus, we can justify at least three separate processes for the four layers: a naming process, a cache process, and a logical-physical process. However, every separate process introduces context switching and communication overhead. The overhead is especially high for layers above the cache layer, which will trigger this overhead for every object operation, including the operations that can be served by the page cache. This then raises the next question: can we justify going beyond three processes?

As we demonstrated in Chapter 3, from a process crash recovery point of view, there is no compelling reason to split up the logical and physical layers into separate processes, or the physical modules from each other. After all, if any of these processes crash, the recovery procedure simply restarts all of them. While it is theoretically possible to recover one of the processes without restarting all others, such an approach would always introduce more complexity–if nothing else, because it would deviate from the normal storage stack startup procedure, thus requiring more code that is used exclusively for recovery purposes. The finer-grained split does help in limiting the damage that any single module can do as a result of software bugs. For example, one physical module cannot overwrite the contents of a device managed by another physical module, and an attempt to do so may be used as indication that the process is misbehaving. There are also smaller general benefits, such as more of the address space being unmapped in each individual process, with the result that faulty, random address accesses are more likely to trigger an exception rather than go undetected. These are however not direct benefits for the recovery system.

Additionally, we argued in Chapter 7 that splitting off the object virtualization

support from the cache layer into its own process would not make process crash recovery less complex. It would help in protecting the cache layer from the virtualization code, but at the cost of performance loss due to the presence of another layer in the path of operations between applications and the page cache. Therefore, this case presents a clear tradeoff between reliability and performance. In Chapter 7, we chose to retain performance.

These cases can be extrapolated to more general principles. Although it is possible that simper solutions apply to specific cases, any (stateful) layer between the cache layer and the device driver layer can always be made part of the infrastructure for checkpoint recovery and cache-log replay presented in Chapter 3, and thus be restarted along with all the other processes in these layers if any of the processes crashes. Thus, for process crash recovery purposes, introducing more processes in the lower layers is not directly beneficial, even if it may be beneficial indirectly for the reasons mentioned above. Adding processes in this part of the stack may have a negative performance impact, but the effects will be limited by the fact that they would be positioned below the cache. In contrast, any stateful layer between the VFS layer and the cache layer could use the transaction support from Chapter 5 to allow for process crash recovery, allowing its process to be restarted individually after a crash. Thus, between the VFS and cache layers, a more fine-grained split-up of the layers may be beneficial, but may come at a steep performance cost due to being positioned in between the application and the page cache. The actual impact depends heavily on the cost of context switching, and thus, different decisions may be appropriate on different platforms.

Besides the four core Loris layers, there are additional possibilities to optimize for performance. For example, in a virtualized environment, the VFS and naming layers can be merged into a single process, as hinted at in Chapter 6. In the resulting situation, the naming layer can no longer be recovered individually, but a naming-layer failure would still result in the shutdown of only the containing virtual environment. On the other side of the stack, each physical module could be merged with the driver for its underlying device. This merge would come at no direct loss of reliability guarantees, as long as no driver is shared between multiple physical modules.

9.1.4 Robustness against software bugs

In Chapters 3, 4, and 5, we have presented three different process crash recovery solutions. These solutions are not equally *robust*, in that they make different assumptions and offer different guarantees. We can now zoom out and look at the bigger picture, which allows us to compare the robustness of the solutions using a loose formalization of the concepts shared between all the layers' crash recovery solutions.

First of all, we assume the presence of a monitoring system, whose purpose is to detect process crashes. As we have noted in various places, we use the term "crash"

as an umbrella term for all detected failures. On MINIX 3, this may be a CPU-generated exception, for example due to execution of a privileged CPU instruction or an illegal memory access; the process itself raising a failure, for example with `panic`, `abort`, `exit`, or a failing `assert`; misbehavior as flagged by the kernel or another process, as a result of IPC interaction; or, the process not responding to heartbeat messages, typically due to being stuck in an endless loop. In all cases, the system process is terminated, and the recovery system kicks in.

When a software bug triggers, the corresponding process may not always crash immediately. This leads to the notion of a *detection interval*: the interval of program execution between the point that a software bug causes program execution to start deviating from the program's intended behavior, and the point that this misbehavior is detected as a crash by the monitoring system. The interval could be measured in transitions (i.e., executed program instructions) or execution time; the choice is not relevant for the discussion. In any case, a detection interval of zero means that a software bug causes immediate failure, which is what we have referred to as a "fail-stop" process crash throughout this work. If no failure is ever observed by the monitoring system, the detection interval is not defined and may be considered infinite. Shortening the detection interval has not been the primary focus of our work; more about this in Sec. 9.2.1.

Once a process has crashed, the corresponding recovery system is responsible for restarting it in an application-transparent manner. As described in Chapter 7, the general approach of our recovery systems is to first roll back to a previous process state, and then roll forward to the expected current process state by replaying previous and/or ongoing requests. This leads to the notion of a *recovery window*: the window of program execution which the recovery system is designed to roll back, thereby returning to a program state equal to, or at least semantically equivalent to, the program's state at the start of the window.

However, even though the recovery window boundaries are conceptually predefined in our work, successful rollback may not always be possible in every window: a software bug may cause misbehavior which either goes beyond the scope of the recovery system (e.g., due to propagation of corrupted results to other components), or which destroys state necessary for successful rollback (e.g., the Dirty State Store in the cache layer). In these cases, we say that the recovery window becomes *invalid*. Unfortunately, there are always forms of misbehavior which are impossible to detect–we have referred to them as *semantic* failures. Thus, the recovery system is not always able to determine whether the recovery window is valid or not, and may therefore have to assume that the window is valid, thus risking recovery to an invalid state.

With these defined terms, we can now state the main characteristic of our recovery systems: *successful recovery* is guaranteed only if detection interval of a triggered software bug is contained entirely in a recovery window, and if this recovery window is valid. Only in that case can the recovery system roll back execution to a point before the software bug triggered, thus discarding any state that may have

been corrupted by the effects of the software bug. At that point, a clean retry of execution can be attempted.

Based on this statement, we can make some qualitative assessments. From the perspective of the recovery system, every triggering software bug has its own associated detection interval, and thus, the detection intervals are a product of circumstances beyond the control of the recovery system. Moreover, we do not know the real-life distribution of detection intervals, although both research by others (e.g., [51]) and our own experience suggest that detection intervals can vary wildly. However, a relatively larger recovery window covers a relatively larger part of this unknown distribution of detection intervals. In other words: without knowing the distribution of bugs' detection intervals, a relatively larger recovery window increases the *probability* that the recovery system can recover from more bugs. In the worst case, a larger recovery window simply has no benefits, but the real-world "Stack overrun" and "Heap corruption" example bugs from Chapter 3 already show that there is a real difference here: without a recovery window that spans multiple requests (as opposed to the single-request recovery windows found in the other layers; more about this below), the recovery system from Chapter 3 would not have been able to recover from these bugs. Thus, the size of the recovery window has an effect on the robustness of a single process's recovery system: a relatively larger recovery window is certainly not worse, and *probabilistically* better.

In order to compare robustness of recovery systems *between* the processes of our storage stack, we need to make the additional assumption that the distribution of bug detection intervals is roughly the same across the processes in the storage stack. We have no data to support (or disprove) this; we are not aware of literature that definitively addresses this point, and a proper investigation into it would be far from trivial. However, we believe the assumption is reasonable, based on the fact that the layers' processes are rather similar: all of the layers have similar request-oriented implementations, with a fair amount of use and manipulation of local data structures, nonpreemptive multithreading, calls into lower layers, and a fair amount of shared library code. Thus, one can expect that the bugs in such behaviorally similar code are also behaviorally similar. With this additional assumption, we can extend the comparison of recovery window sizes to span multiple processes, and state that across the Loris layers, a recovery system with a larger recovery window size can be said to offer stronger guarantees for recovery, and thus reasonably be deemed *more robust*, at least in theory. This is our first point of comparison.

In addition, for successful recovery from a triggered software bug, the recovery window must be valid. Again, the validity of the recovery window itself is beyond the control of the recovery system. In fact, it is impossible to rule out the possibility of semantic failures, and thus, there will always be cases where the recovery system wrongly concludes that recovery is possible. However, similarly to the argument above, if there are fewer ways in which the recovery window can be invalidated, then any occurring bug is *probabilistically* less likely to invalidate the recovery window. Thus, for a single process's recovery system, and with all other things being equal,

having to make fewer assumptions about the validity of the recovery window is certainly not worse, and *probabilistically* better. Comparing assumptions *between* processes would be impossible however, were it not for the fact that between our Loris recovery systems, the sets of assumptions are strict subsets or supersets of one another. In addition to the previous assumptions, this fact allows us to state that a recovery system with fewer assumptions about the validity of its recovery windows offers stronger guarantees for recovery, and thus reasonably be deemed (again, theoretically) *more robust*. This is our second point of comparison. We note that while there is a possible conflict between the two points of comparison, it will become apparent that this is not an issue when comparing our recovery systems.

Finally, one redeeming factor for a recovery system is whether it can detect certain invalidations of its recovery windows. If, after crash, the recovery system can determine that the recovery window is invalid, it can avoid rolling back and attempting recovery anyway, instead informing the system administrator that an irrecoverable failure has occurred and leaving any further recovery attempts to other actors. Again, given the existence of semantic failures, the recovery system can possibly perform such detection only for a certain subset of recovery window invalidations.

Summarizing, we can compare recovery systems using two main characteristics:

- The size of the recovery window (the larger, the better);
- The assumptions made about the validity of the recovery window (the fewer, the better).

In addition, as a secondary point of interest, the following is also worth considering:

- Which recovery window invalidations can be detected (the more, the better).

Based on these definitions and the previously stated assumptions, we can conclude that the lower-layer process recovery system from Chapter 3 offers by far the strongest guarantees. Its recovery windows are delineated by **sync** calls, which each establish a new consistency point and clear the cache-layer roll-forward log. As a result, the recovery window typically spans several seconds. In addition, there is a minimal set of assumptions regarding the validity of the recovery window. Not only may any misbehavior occur within the affected process itself, but due to the "blunt" recovery approach of restarting of all lower-layer processes after a crash, propagation of failures *between* these two layers will not jeopardize recovery either. A physical layer may even write to disk blocks which are not part of the consistency point to which the system will roll back during recovery.

In contrast, the cache-layer process recovery system from Chapter 4 offers the weakest guarantees. Its recovery window is delineated by the request processing cycle, and thus, internal propagation of corrupted state across requests may prevent successful recovery. Moreover, there are many more assumptions about the validity

of the recovery window: not only may corrupted state not be propagated across process boundaries in either direction (to either upper or lower layers), it may also not propagate into any parts of the Dirty State Store (DSS). The DSS checksums do detect recovery window invalidations resulting from wild writes that affect DSS memory, but do not protect against corrupted calls into the DSS. Neither (modifying) object operations sent down to the lower layers, nor updates to the DSS are deferred until the end of each request, and therefore, there are various scenarios where the recovery window may become invalid. We have discussed a possible improvement at this point in Chapter 8.

Finally, the naming-layer process recovery system from Chapter 5 has the same delineation of the recovery window as for the cache layer, namely requests. However, compared to the cache-layer recovery system, the naming-layer recovery system has fewer cases where the recovery window is invalidated, for two reasons. First, no state needs to be recovered from the crashed process's image, and thus, no form of internal state corruption can cause recovery to fail. Second, each request's modifying object operations are bundled in a single transaction which is sent down to the cache layer only at the very end of processing the request. Thus, the naming layer cannot perform actions in the middle of processing a request that the recovery system cannot undo. Thus, even though the naming-layer recovery system has a smaller recovery window than the lower-layer recovery system *and* a superset of its assumptions, it does have a subset of the assumptions made by the cache-layer recovery system.

Summarizing, the lower-layer process recovery system is by far the most robust, followed by the naming-layer system, with the cache-layer system being the least robust. We believe that these levels of robustness correspond well to the expected distribution of software bugs across the respective layers. Combined, the logical and physical layers form by far the most complex part of the stack, and many future extensions that further increase complexity can be expected in these layers. Especially extensions that involve background activity (e.g., a log-structured layout in the physical layer [112], or background object migration in the logical layer) clearly benefit from recovery windows that extend beyond a single request. As we argued in Chapter 5, the risk of the naming layer lies more in issues that come up while processing a single request, such as complex parsing of file formats. Finally, the cache layer is conceptually the simplest layer and thus theoretically the least likely to be affected by software bugs at all. Therefore, it is arguably the layer where investing more resources into providing stronger recovery guarantees is least warranted. However, there is no denying that the virtualization support added in Chapter 7 has made the cache layer more complex. The same chapter has suggested two possible approaches to mitigate this new problem.

We fully acknowledge that our robustness model is based on theory rather than practice. Given that the random fault injection experiments in Chapter 7 do not take into account the assumptions made by the respective recovery systems, one could expect that the layers with the smaller subsets of assumptions would see relatively

more successful recoveries. However, the results of the experiments do not show a significant difference in this respect. One possible explanation is the fact that we had to inject many random faults at once in order to trigger a reasonable number of failures within the available experimentation time, which could have led to insufficient testing of the recovery window boundaries. More realistic fault injection methods are still subject of ongoing research (e.g., [147]).

9.2 Future work

In addition to the technical limitations we mentioned in Chapter 8, we believe that our work leaves several avenues for subsequent work. In this section, we describe four main areas of future work: software bugs (Sec. 9.2.1), virtualization (Sec. 9.2.2), performance (Sec. 9.2.3), and emerging technologies (Sec. 9.2.4).

9.2.1 Software bugs

As mentioned in Sec. 9.1.4, our solutions for recovery from process crashes depend on the detection that a fault has occurred at all. Thus, the overall resilience to software bugs improves with the ability to detect failures, and detect them as quickly as possible. One way to do that, as we have suggested in Chapter 4, is to perform more `assert`-type checks at run time in order to detect any unexpected behavior as early as possible.

A particularly interesting place for more consistency checks is the boundary between the layers. Each module could perform checks on the incoming requests and replies from other modules. If the checks do not match the receiver's expectations, it could flag the sender as having crashed. We have experimented with this concept on MINIX 3 at the driver level before [65], and others have done research on more high-level boundary checks underneath file systems [41]. We believe that similar checks in the Loris storage stack could greatly benefit the overall effectiveness of our software bug solutions. It would be worthwhile to investigate which forms of misbehavior can be detected at each of the layers, especially considering that this form of detection is always active and thus must have low overhead, even aside from the ever-present risk that the extra checks introduce new bugs themselves. We consider this one of the primary areas of future work.

However, there are always classes of software bugs which result in failures that can practically not be tested. This applies in particular to *semantic* failures, which result in seemingly correct but unintended behavior. We contend that in the space of low-cost solutions that we have been exploring in this work, there is no reasonable solution for this problem. One possible approach is N-version programming [13], but when applied in the storage stack, this approach comes at a high cost and with severe limitations [17].

Aside from improving detection, it is worth considering whether and how recovery can be improved. Such improvements would involve any of the points affecting

the recovery window as discussed in Sec. 9.1.4. Extending the size of the recovery window (the first point) is easy in the lower layers, as it merely requires a different configuration, with a less frequent checkpointing interval and consequently more memory spent on the roll-forward log. For the naming and cache layers however, extending the recovery window beyond single requests would not only require a similar roll-forward log in the layer above the affected layer–thus spending extra memory which, unlike within the cache layer, cannot be reused for other purposes–but also impose more determinism in these two layers, since pre-crash results have already been returned to applications and thus may not change during replay.

With respect to the validity of the recovery window (the second point), the cache layer makes the most assumptions and may thus benefit the most from improvements at this point. However, the current situation could be improved substantially only by providing better isolation for the Dirty State Store, for example by moving it into a separate process, which would be prohibitively expensive in terms of extra context switching and memory copying overhead. In general, we find that there are few cases where invalidation of the recovery window can be determined after the fact (the third point), without also possibly serving as a consistency check for immediate detection. In that sense, the Dirty State Store appears to be somewhat of an atypical case.

Given that process crash recovery has been a major focus of this thesis, we believe that within the scope of our principles and goals as stated in the general introduction, there are few substantial improvements to be made in this area when continuing the path we have taken. Instead, as we have mentioned, we believe that a fruitful direction of future work is the combination of specific and generic recovery techniques, thereby combining the advantages of high-level knowledge specific to the storage stack with the simplicity of automated generation of most of the infrastructure needed for verification and recovery.

9.2.2 Virtualization

It is clear that Chapters 6 and 7 have not provided an exhaustive exploration of our virtualization concept. In fact, some of the assertions in Chapter 6 have not yet been tested in an implementation. This has been mainly a result of time constraints, although many of the virtualization aspects fall well outside the scope of reliability in general and this thesis in particular. So far, our implementation has been successful for the reliability aspects of our virtualization work: even though the results were not sufficiently interesting to report in the earlier chapters, the handling of crashes in the virtual domains (e.g. of the VFS process) has shown to work exactly as intended. In addition, our implementation is a successful, if basic, proof of concept for the virtualization aspects related to the storage stack. However, we do believe that several additional aspects of our new virtualization approach are worth exploring further, and we consider this to be one of the main areas of future work.

Most notably, we have not performed a direct performance comparison with

other virtualization approaches such as Xen [19]. Since MINIX 3 is not optimized for low-level performance, we believe that such a comparison would not have helped in exploring the potential of the concept. It would likely have shown that MINIX 3 itself has a large relative performance overhead, instead of providing any insight into the performance of the virtualization aspects themselves. Optimizing MINIX 3 to be performance-competitive with Xen would not be a trivial task. Thus, a proper performance evaluation may require a different base platform, such as L4 [96] for a microkernel implementation, and perhaps even a commodity operating system such as Linux for a monolithic implementation. Needless to say, reimplementing our ideas, and in particular our storage stack, on top of these platforms would not be trivial either.

Even though the storage side of the virtualization prototype has received most of our attention, there are still important missing features there as well. One major example is subobject copy-on-write and deduplication. Sharing of storage for parts of objects between domains would be highly beneficial for files that are large and have only part of their contents modified in-place by the individual domains. One typical example of such files is a virtual hard disk image [102], but there are other examples of files which are perhaps more likely to be used in our virtualized environments, such as package management database files. In our storage stack, subobject copy-on-write support could be implemented entirely in separate physical modules, by implementing different behavior for the `copy` operation. However, this would not allow individual pages to be shared in memory, as that would require involvement of the cache layer. Subobject deduplication would similarly benefit from involving the cache layer, and an optimal subobject sharing implementation may in fact require involvement of several layers.

Another example is full sharing, rather than copy-on-write sharing, of part of the file system hierarchy between virtual domains. Full sharing of a subset of files would allow applications in separate domains to cooperate more closely, without the need for a local networked file system. This concept was proposed for a storage virtualization approach similar to ours [72]. While we believe that full sharing is not needed in the majority of use cases for our virtualization approach, it would be interesting to see if and how full sharing could be made to work in our storage stack. Our support for transactions is already one step in the right direction, but while our naming layer is stateless in terms of cached *modifications*, it is not free of cached *read-only* state. The main remaining challenge is synchronizing cached state between domains while avoiding a corresponding performance hit.

In addition, we have not experimented with a number of largely orthogonal features that are expected to be offered by any contemporary virtualization solution, including resource isolation, checkpointing, and migration. We believe that designing a complete virtualization solution that incorporates all these features while also retaining high performance would be a considerable research challenge.

9.2.3 Performance

In the general introduction of this dissertation, we have stated that we consider fine-grained modularity and isolation to be one of the pillars on which we base our research. This choice has allowed us to provide stronger reliability guarantees in Chapters 3, 4, and 5 than in similar contemporary work, and has shown to be a precondition for our virtualization approach in Chapter 6. However, as shown in the experiments of Chapter 2, the process separation itself does come at a considerable cost. Thus, our work would directly benefit from further research into performance optimizations for the storage stack, as long as these optimizations retain the isolation guarantees on which we have relied for our solutions.

One possible improvement would be the use of shared memory between the components. Right now, only a minimal amount of information is contained in the messages that are passed using interprocess communication (IPC), since in the MINIX 3 version on which we based our work, these messages are only 36 bytes in size. Any additional data must be transferred using capability-like memory grants [62]. In order to maintain isolation between processes, copying data from or to a memory grant must be done by the kernel, and thus, each interprocess copy requires an additional kernel call. These kernel calls could be eliminated by preestablishing shared-memory regions between pairs of processes, mapped in read-write by the sender, and read-only by the receiver of data. Moreover, memory for data could be shared between more than two processes, thereby eliminating the need for extra copies altogether in some cases. These concepts have been tried successfully on the MINIX 3 network stack [67].

Even without such modifications, the Loris prototype already avoids many unnecessary memory copies. For example, the naming and logical layers do not copy in data unless they have to, instead letting the cache and physical layers copy directly from and to applications and each other, using forwarded (*indirect*) memory grants. However, as it is, data blocks are copied from and to the physical layer, so that this layer can generate and verify checksums for these data blocks. In theory, it would be possible to eliminate the extra copies by letting the device read and write from and to cache pages directly. In that case, checksum generation for data pages (for `write` operations) could be left entirely to the cache layer, using the checksum propagation introduced in Chapter 5. Checksum verification of data pages (for `read` operations) still must be done by the physical modules, since the logical layer must be informed about verification failures immediately; the physical modules could be given read-only shared memory access to the appropriate cache pages for this purpose.

While use of shared memory would mainly serve to improve the performance of the baseline in our work, several aspects of our subsequent reliability work benefit from such changes in similar ways. For example, shared memory would allow for further overhead reduction for propagation of checksums through the stack (see Chapter 4), and eliminate the overhead for the problematic case of subpage writes in transactions (see Chapter 7).

The remaining question is whether the use of shared memory would create complications for our reliability improvements. Even though read-only mappings prevent a receiver from modifying the state of the sender, the sender could modify the shared content while the receiver is accessing it, possibly leading to new, subtle *time of check to time of use* issues. More research and experiments are necessary to measure not only the performance improvement, but also the overall impact on reliability of a fullblown shared-memory solution.

The work on the MINIX 3 network stack also experimented with dedicating processor cores to system processes, in order to reduce context switching overhead [67]. Once communication based on shared memory is in place, the same could be tried on our storage stack. We expect that in order to get the maximum yield out of these changes, the operation of the Loris prototype would have to be made more asynchronous (e.g., with proper background flushing in the cache layer), so as to prevent that processes (and thus cores) spend most of their time waiting for each other. Compared to the network stack, the storage stack poses new challenges for asynchrony. For example, there is a fundamental difference between read and write operations in the storage stack. Reads must be served immediately, whereas processing of writes may be deferred. Increased asynchrony may also have a further, currently unknown impact on reliability: even though we believe our reliability solutions would continue to work, the asynchrony may have a negative impact on their effectiveness, for example by further shortening or invalidating recovery windows. All of these areas require further research.

9.2.4 Emerging technologies

Our work has focused on improving reliability for currently available technologies, thus allowing our solutions to be used on systems that exist today. As new technologies emerge and are adopted for use in new systems, our reliability solutions may have to be reevaluated accordingly.

For new storage technologies in particular, the design of Loris allows the details of a storage device to be hidden entirely in its corresponding physical module. Storage devices that provide an interface compatible with the interface used for hard disks, as is the case for Solid State Drives (SSDs), could use our generic physical modules. However, specific optimizations for such new devices may warrant the use of different layouts and thus the implementation of separate physical modules. Similarly, it should be entirely possible to add support for devices with different interfaces, such as “raw” flash memory (without a translation layer) and even object storage devices [6], in the physical layer alone.

The use of new devices for a purpose other than generic storage of objects presents a slightly bigger challenge. For example, one might want to dedicate an entire SSD to storing the deduplication index [100]. In that case, involvement of the logical layer would be unnecessary and require specific exceptions, in particular because the dedicated SSD should not contain a copy of other stack metadata. It might help

to maintain the SSD contents through a custom physical module, used by the cache layer directly. Such a change would then also require a new look at the problems of storage device failures, whole-system failures, and software bugs, although we expect that many of our solutions can be adapted to this new situation quite easily.

When it comes to emerging technologies that affect the storage stack, the biggest potential game changer is Non-Volatile Memory (NVM), for example Phase Change Memory (PCM) [120] and Spin-Transfer Torque RAM (STT-RAM) [78]. NVM is byte accessible, random access, nonvolatile, and only slightly slower than DRAM [92]. As a result, NVM can be used as both working memory and permanent storage at the same time. The (future) availability of NVM, either instead of DRAM or in addition to it, poses a major challenge in system design. Various ideas have been proposed for integration of such memory into file system design [30], the operating system interface [157], and whole-system design [105].

Even if we were to limit the use of NVM to just Loris, there are several ways to make use of such memory. The least disruptive approach would be to manage the system's NVM in a separate physical module, or to use NVM to help manage consistency points within an existing physical module, although it then becomes unclear whether a page cache is still necessary. More invasive changes would likely allow for more optimal use of the memory, in particular with respect to dealing with whole-system failures–for example, the NVM could be used for the page cache, or at least the cache log from Chapter 3. We believe that once there is more clarity regarding the exact hardware properties and prices of NVM, many interesting avenues of research into storage stack reliability will open up.

References

[1] Slicing-by-8. `http://slicing-by-8.sourceforge.net/`.

[2] FUSE: Filesystem in userspace. `http://fuse.sourceforge.net/`.

[3] Filesystems in userspace: puffs, refuse, FUSE, and more. `http://www.netbsd.org/docs/puffs/`.

[4] GNU Hurd – subhurds. `http://www.gnu.org/software/hurd/hurd/subhurd.html`.

[5] *System Application Program Interface (API) [C Language]: IEEE Std 1003.1-1990 (Revision of IEEE Std 1003.1-1988)*. Information Technology - Portable Operating System Interface. IEEE, 1990. ISBN 978-1559370615.

[6] Object-based storage device commands, ANSI standard INCITS 400-2004, 2004.

[7] Raja Appuswamy. *Building a File-Based Storage Stack: Modularity and Flexibility in Loris*. PhD thesis, 2014.

[8] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Block-level RAID is dead. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage 2010. USENIX Association, 2010.

[9] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Loris - a dependable, modular file-based storage stack. In *Dependable Computing, 2010 IEEE 16th Pacific Rim International Symposium on*, PRDC 2010, pages 165–174. IEEE Computer Society, 2010.

[10] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum.

Loris - a dependable, modular file-based storage stack. Technical Report IR-CS-61, Department of Computer Science, Vrije Universiteit, Amsterdam, 2010.

[11] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Flexible, modular file volume virtualization in Loris. In *Mass Storage Systems and Technologies, 2011 IEEE 27th Symposium on*, MSST 2011, pages 1–14. IEEE Computer Society, 2011.

[12] Ken Arnold and James Gosling. *The Java Language Specification*. Addison-Wesley Professional, 2000. ISBN 978-0201634556.

[13] Algirdas Avizienis and Liming Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings the First IEEE-CS International Computer Software and Applications Conference*, COMPSAC 1977, pages 149–155. IEEE Computer Society, 1977.

[14] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[15] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an object store. In *Mass Storage Systems and Technologies, 2003. Proceedings. 20th IEEE/11th NASA Goddard Conference on*, MSST 2003, pages 165–176. IEEE Computer Society, 2003.

[16] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies*, FAST 2008. USENIX Association, 2008.

[17] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating file-system mistakes with EnvyFS. In *Proceedings of the 2009 USENIX Annual Technical Conference*, USENIX ATC 2009. USENIX Association, 2009.

[18] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP 1991, pages 198–212. ACM, 1991.

[19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP 2003, pages 164–177. ACM, 2003.

[20] Aaron Brown and David A. Patterson. Towards availability benchmarks: A case study of software RAID systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, USENIX ATC 2000, pages 263–276. USENIX Association, 2000.

[21] Olaf Buddenhagen. Advanced lightweight virtualization. http://triceps.blogspot.com/2007/10/advanced-lightweight-virtualization.html, 2007.

[22] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[23] Thomas Bushnell. Towards a new strategy of OS design. *GNU's Bulletin*, 1 (16), 1994.

[24] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI 2004. USENIX Association, 2004.

[25] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *Parallel and Distributed Processing with Applications, 2011 IEEE 9th International Symposium on*, ISPA 2011, pages 244–249. IEEE Computer Society, 2011.

[26] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HotOS 2001, pages 133–138. USENIX Association, 2001.

[27] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 74–83. ACM, 1996.

[28] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI 2005, pages 273–286. USENIX Association, 2005.

[29] Cluster File Systems, Inc. Lustre: A scalable, high performance file system. 2002.

[30] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP 2009, pages 133–146.

ACM, 2009.

[31] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI 2008, pages 59–72. USENIX Association, 2008.

[32] Timothy J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. 1997.

[33] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the 2002 USENIX Annual Technical Conference*, USENIX ATC 2002, pages 177–190. USENIX Association, 2002.

[34] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Journal-guided resynchronization for software RAID. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*, FAST 2005. USENIX Association, 2005.

[35] Dave Dopson. SoftECC: A system for software memory integrity checking. Master's thesis, Massachusetts Institute of Technology, 2005.

[36] Hideki Eiraku, Yasushi Shinjo, Calton Pu, Younggyun Koh, and Kazuhiko Kato. Fast networking with socket-outsourcing in hosted virtual machine environments. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 310–317. ACM, 2009.

[37] Akira Eto, Mitsumori Hidaka, Yutaka Okuyama, Katsutaka Kimura, and Masayuki Hosono. Impact of neutron flux on soft errors in MOS memories. In *Electron Devices Meeting, 1998. Technical Digest., International*, IEDM 1998, pages 367–370. IEEE, 1998.

[38] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. In *Proceedings of the 2005 IEEE International Symposium on Mass Storage Systems and Technology*, LGDI 2005, pages 119–123. IEEE Computer Society, 2005.

[39] David Fiala, Kurt B. Ferreira, Frank Mueller, and Christian Engelmann. A tunable, software-based DRAM error detection and correction library for HPC. In *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, EuroPar 2011, pages 251–261. Springer-Verlag, 2011.

[40] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI 1996, pages 137–151. ACM, 1996.

[41] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies*, FAST 2012. USENIX Association, 2012.

[42] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[43] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the First USENIX Conference on Operating Systems Design and Implementation*, OSDI 1994. USENIX Association, 1994.

[44] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, NDSS 2003, pages 191–206. USENIX Association, 2003.

[45] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop*, EW 9, pages 109–114. ACM, 2000.

[46] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 92–103. ACM, 1998.

[47] Cristiano Giuffrida and Andrew S. Tanenbaum. Cooperative update: A new model for dependable live update. In *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, HotSWUp 2009, pages 1–6. ACM, 2009.

[48] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. We crashed, now what? In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep 2010, pages 1–8. USENIX Association, 2010.

[49] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer*, 7 (6):34–45, 1974.

[50] Robert P. Goldberg and Robert Hassinger. The double paging anomaly. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*, pages 195–199. ACM, 1974.

[51] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux kernel behavior under errors. In *Dependable Sys-*

tems and Networks, 2003. Proceedings. 2003 International Conference on, DSN 2003, pages 459–468. IEEE Computer Society, 2003.

[52] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware 2006, pages 342–362. Springer-Verlag, 2006.

[53] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP 1987, pages 155–162. ACM, 1987.

[54] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP 2011, pages 71–83. ACM, 2011.

[55] Hermann Härtig. Security architectures revisited. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 16–23. ACM, 2002.

[56] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP 1997, pages 66–77. ACM, 1997.

[57] Les Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, 1997.

[58] HDF Group, The. Hierarchical data format version 5, 2000–2010. `http://www.hdfgroup.org/HDF5`.

[59] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994.

[60] Gernot Heiser and Ben Leslie. The OKL4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the First ACM Asia-Pacific Workshop on Systems*, APSYS 2010, pages 19–24. ACM, 2010.

[61] Val Henson and Theodore Ts'o. Double the metadata, double the fun: A cow-like approach to file system consistency. `http://valerieaurora.org/review/doublefs.pdf`.

[62] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a highly dependable operating system. In *Proceedings of the Sixth European Dependable Computing Conference*, EDCC 2006, pages 3–12. IEEE Computer Society, 2006.

[63] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *Dependable Systems and Networks, 2007. 37th Annual IEEE/IFIP International Conference on*, DSN 2007, pages 41–50. IEEE Computer Society, 2007.

[64] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault isolation for device drivers. In *Dependable Systems Networks, 2009. IEEE/IFIP International Conference on*, DSN 2009, pages 33–42. IEEE Computer Society, 2009.

[65] Jorrit N. Herder, David C. van Moolenbroek, Raja Appuswamy, Bingzheng Wu, Ben Gras, and Andrew S. Tanenbaum. Dealing with driver failures in the storage stack. In *Proceedings of the 2009 Fourth Latin-American Symposium on Dependable Computing*, LADC 2009, pages 119–126. IEEE Computer Society, 2009.

[66] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC 1994. USENIX Association, 1994.

[67] Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S Tanenbaum. Keep net working - on a dependable and fast networking stack. In *Dependable Systems and Networks, 2012 42nd Annual IEEE/IFIP International Conference on*, DSN 2012, pages 1–12. IEEE Computer Society, 2012.

[68] Thomas Huckle. Collection of software bugs. `http://www5.in.tum.de/~huckle/bugse.html`, 2015.

[69] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 111–122. ACM, 2012.

[70] Intel Corporation. Intel SSE4 programming reference. 2007.

[71] Intel Corporation. Fast CRC computation for iSCSI polynomial using CRC32 instruction. 2011.

[72] William Jannen, Chia-Che Tsai, and Donald E. Porter. Virtualize storage, not disks. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS 2013. USENIX Association, 2013.

[73] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 14–24, 2006.

[74] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: A file system for virtualized flash storage. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies*, FAST 2010. USENIX Association, 2010.

[75] Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok. RAIF: Redundant array of independent filesystems. In *Proceedings of Twenty-Fourth IEEE Conference on Mass Storage Systems and Technologies*, MSST 2007, pages 199–212. IEEE Computer Society, 2007.

[76] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, 2000.

[77] Wei-Lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, 1993.

[78] Takayuki Kawahara. Scalable spin-transfer torque RAM technology for normally-off computing. *Design Test of Computers, IEEE*, 28(1):52–63, 2011.

[79] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST 2004, pages 59–62. USENIX Association, 2004.

[80] Peter Kelemen. Silent corruptions. In *8th Annual Workshop on Linux Clusters for Super Computing*, 2007.

[81] Cheryl Kemp. Data loss causes companies to hemorrhage $1.7 trillion per year: Report. `http://www.thewhir.com/web-hosting-news/data-loss-causes-companies-hemorrhage-1-7-trillion-per-year-report`, 2014.

[82] Dongsung Kim, Hwanju Kim, Myeongjae Jeon, Euiseong Seo, and Joonwon Lee. Guest-aware priority-based virtual machine scheduling for highly consolidated server. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, EuroPar 2008, pages 285–294. Springer-Verlag, 2008.

[83] Hwanju Kim, Heeseung Jo, and Joonwon Lee. XHive: Efficient cooperative caching for virtual machines. *IEEE Transactions on Computers*, 60(1):106–119, 2011.

[84] Steven R Kleiman. Vnodes: An architecture for multiple file system types

in Sun UNIX. In *Proceedings of the USENIX Summer 1986 Conference*, volume 86, pages 238–247. USENIX Association, 1986.

[85] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. Technical report, Aalborg University, 2007.

[86] Kevin Klues, Barret Rhoden, David Zhu, Andrew Waterman, and Eric Brewer. Processes and resource management in a scalable many-core OS. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, HotPar 2010. USENIX Association, 2010.

[87] Philip Koopman. 32-bit cyclic redundancy codes for internet applications. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN 2002, pages 459–472. IEEE Computer Society, 2002.

[88] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. Parity lost and parity regained. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies*, FAST 2008, pages 1–15. USENIX Association, 2008.

[89] Adam Lackorzyński, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT 2012, pages 93–102. ACM, 2012.

[90] Butler W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer, 1980.

[91] John R. Lange and Peter Dinda. SymCall: Symbiotic virtualization through VMM-to-guest upcalls. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2011, pages 193–204. ACM, 2011.

[92] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA 2009, pages 2–13. ACM, 2009.

[93] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery domains: An organizing principle for recoverable operating systems. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 49–60. ACM, 2009.

[94] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and

implementation of an operating system to support distributed multimedia applications. *Selected Areas in Communications, IEEE Journal on*, 14(7):1280–1297, 1996.

[95] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, USENIX ATC 2007, pages 1–6. USENIX Association, 2007.

[96] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP 1995, pages 237–250. ACM, 1995.

[97] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST 2013, pages 31–44. USENIX Association, 2013.

[98] Pin Lu and Kai Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *Proceedings of the 2007 USENIX Annual Technical Conference*, USENIX ATC 2007, pages 29–43. USENIX Association, 2007.

[99] Dan Magenheimer, Chris Mason, Dave McCracken, and Kurt Hackel. Paravirtualized paging. In *Proceedings of the First Conference on I/O Virtualization*, WIOV 2008. USENIX Association, 2008.

[100] Dirk Meister and André Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Mass Storage Systems and Technologies, 2010 IEEE 26th Symposium on*, MSST 2010. IEEE Computer Society, 2010.

[101] Alan Messer, Philippe Bernadat, Guangrui Fu, Deqing Chen, Zoran Dimitrijevic, David Lie, Durga Devi Mannaru, Alma Riska, and Dejan Milojicic. Susceptibility of commodity systems and software to memory soft errors. *IEEE Transactions on Computers*, 53(12):1557–1568, 2004.

[102] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST 2011. USENIX Association, 2011.

[103] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC 2013, pages 279–290. USENIX Association, 2013.

[104] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 USENIX Annual Technical Conference*, USENIX ATC 2009. USENIX Association, 2009.

[105] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[106] Ruslan Nikolaev and Godmar Back. VirtuOS: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP 2013, pages 116–132. ACM, 2013.

[107] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, 1996.

[108] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, 40(1):41–50, 1996.

[109] Open Group, The. fsync – the open group base specifications issue 7. `http://pubs.opengroup.org/onlinepubs/9699919799/functions/fsync.html`, 2013.

[110] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, OSDI 2002, pages 361–376. USENIX Association, 2002.

[111] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2002, pages 55–64. ACM, 2002.

[112] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23: 11–28, 1989.

[113] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD 1988, pages 109–116. ACM, 1988.

[114] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI 2006. USENIX Association, 2006.

[115] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP

2009, pages 161–176. ACM, 2009.

[116] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP 2005, pages 206–220. ACM, 2005.

[117] Daniel Price and Andrew Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th Large Installation System Administration Conference*, LISA 2004, pages 241–254. USENIX Association, 2004.

[118] Himanshu Raj and Karsten Schwan. O2S2: Enhanced object-based virtualized storage. *ACM SIGOPS Operating Systems Review*, 42(6):24–29, 2008.

[119] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Coerced cache eviction and discreet mode journaling: Dealing with misbehaving disks. In *Dependable Systems & Networks, 2011 IEEE/IFIP 41st International Conference on*, DSN 2011, pages 518–529. IEEE Computer Society, 2011.

[120] S. Raoux, G.W. Burr, M.J. Breitwisch, C.T. Rettner, Y.C. Chen, R.M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.L. Lung, and C.H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[121] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. Hype and virtue. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HotOS 2007. USENIX Association, 2007.

[122] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[123] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Dependable Systems and Networks, 2006. International Conference on*, DSN 2006, pages 249–258. IEEE Computer Society, 2006.

[124] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS 2009, pages 193–204. ACM, 2009.

[125] Margo Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS 2009. USENIX Association, 2009.

[126] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the EROS single-level store. In *Proceedings of the 2002 USENIX Annual Technical Conference*, USENIX ATC 2002, pages 59–72. USENIX Association, 2002.

[127] Philip P. Shirvani, Nirmal R. Saxena, and Edward J. McCluskey. Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49(3):273–284, 2000.

[128] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, StorageSS 2005, pages 26–36. ACM, 2005.

[129] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. *ACM Transactions on Storage*, 1(2):133–170, 2005.

[130] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI 2010, pages 1–8. USENIX Association, 2010.

[131] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the Second ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys 2007, pages 275–287. ACM, 2007.

[132] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, FAST 2009, pages 29–42. USENIX Association, 2009.

[133] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying file system protection. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 79–90. USENIX Association, 2001.

[134] Lex Stein. Stupid file systems are better. In *Proceedings of the Tenth Conference on Hot Topics in Operating Systems*, HotOS 2005. USENIX Association, 2005.

[135] STMicroelectronics. STM32 reference manual. 2011.

[136] Sun Microsystems. Solaris ZFS file storage solution. Solaris 10 data sheets. 2004.

[137] Sun Microsystems. Lustre: End to end data integrity design. 2009.

[138] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale,

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating system support for restartable file systems. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies*, FAST 2010. USENIX Association, 2010.

[139] Swaminathan Sundararaman, Laxman Visampalli, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Refuse to crash with Re-FUSE. In *Proceedings of the Sixth European Conference on Computer Systems*, EuroSys 2011. ACM, 2011.

[140] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP 2003, pages 207–222. ACM, 2003.

[141] Hidekazu Tadokoro, Kenichi Kourai, and Shigeru Chiba. A secure system-wide process scheduler across virtual machines. In *Dependable Computing, 2010 IEEE 16th Pacific Rim International Symposium on*, PRDC 2010, pages 27–36. IEEE Computer Society, 2010.

[142] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (Third Edition)*. Prentice Hall, 2006. ISBN 978-0131429383.

[143] Vasily Tarasov, Deepak Jain, Dean Hildebrand, Renu Tewari, Geoff Kuenning, and Erez Zadok. Improving I/O performance using virtual disk introspection. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage 2013. USENIX Association, 2013.

[144] David Teigland and Heinz Mauelshagen. Volume managers in Linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 185–197. USENIX Association, 2001.

[145] Tezzaron Semiconductor. Soft errors in electronic memory – a white paper, 2004.

[146] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage*, 4(2):5:1–5:56, 2008.

[147] Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. Evaluating distortion in fault injection experiments. In *Proceedings of the 15th IEEE International Symposium on High-Assurance Systems Engineering*, HASE 2014. IEEE Computer Society, 2014.

[148] Richard van Heuven van Staereling, Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Efficient, modular metadata management with Loris. In *Proceedings of the 6th IEEE International Conference on Net-*

working, Architecture and Storage, NAS 2011, pages 278–287. IEEE Computer Society, 2011.

[149] David C. van Moolenbroek, Raja Appuswamy, and Andrew S. Tanenbaum. Integrated end-to-end dependability in the Loris storage stack. In *Proceedings of the Seventh Workshop on Hot Topics in System Dependability*, HotDep 2011. IEEE Computer Society, 2011.

[150] David C. van Moolenbroek, Raja Appuswamy, and Andrew S. Tanenbaum. Integrated system and process crash recovery in the Loris storage stack. In *Networking, Architecture and Storage, 2012 IEEE 7th International Conference on*, NAS 2012, pages 1–10. IEEE Computer Society, 2012.

[151] David C. van Moolenbroek, Raja Appuswamy, and Andrew S. Tanenbaum. Battling bad bits with checksums in the Loris page cache. In *Dependable Computing, 2013 Sixth Latin-American Symposium on*, LADC 2013, pages 68–77. IEEE Computer Society, 2013.

[152] David C. van Moolenbroek, Raja Appuswamy, and Andrew S. Tanenbaum. Transaction-based process crash recovery of file system namespace modules. In *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC 2013, pages 338–347. IEEE Computer Society, 2013.

[153] David C. van Moolenbroek, Raja Appuswamy, and Andrew S. Tanenbaum. Putting the pieces together: The construction of a reliable virtualizing object-based storage stack. In *Proceedings of the Second International Symposium on Computing and Networking Across Practical Development and Theoretical Research*, CANDAR 2014. IEEE Computer Society, 2014.

[154] David C. van Moolenbroek, Raja Appuswamy, and Andrew S. Tanenbaum. Towards a flexible, lightweight virtualization alternative. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, pages 1–7. ACM, 2014.

[155] Tom Van Vleck. Unix and Multics. `http://www.multicians.org/unix.html`, 1993.

[156] Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S. Tanenbaum. Techniques for efficient in-memory checkpointing. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, HotDep 2013, pages 1–5. ACM, 2013.

[157] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104. ACM, 2011.

[158] Carl A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[159] Carsten Weinhold and Hermann Härtig. VPFS: Building a virtual private file system with a small trusted computing base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, EuroSys 2008, pages 81–93. ACM, 2008.

[160] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies*, FAST 2008, pages 1–17. USENIX Association, 2008.

[161] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[162] David A. Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2009.

[163] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, 2002.

[164] Mark Williamson. XenFS, 2009.

[165] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE Computer Society, 2009.

[166] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, and Tzi-cker Chiueh. A feather-weight virtual machine for Windows applications. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE 2006, pages 24–34. ACM, 2006.

[167] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies*, FAST 2010. USENIX Association, 2010.

[168] Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Zettabyte reliability with flexible end-to-end data integrity. In *Mass Storage Systems and Technologies, 2013 IEEE 29th Symposium on*, MSST 2013, pages 1–14. IEEE Computer Society, 2013.

[169] J. F. Ziegler and W. A. Lanford. The effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

Summary

A computer system may experience various problems in its components, both in hardware and in software. These problems generally do not occur frequently, but if they are not properly handled, they may cause the entire computer system to malfunction. Such a malfunction may not only interrupt the system's regular operation, but also cause data loss, and ultimately even result in financial loss to the owner of the computer system. In order to avoid larger malfunctions, one can improve the computer system's ability to deal with the problems in individual components, thereby making the system more **reliable**.

The context of our work is research on improving computer systems reliability through modifications in software. Our general working area is the *operating system*, typically the lowest layer in the software stack. One of the operating system's most important tasks is to handle hardware interaction on behalf of the applications running on top of the operating system. This makes the operating system particularly important for reliability: not only is the operating system the first software layer to be exposed to problems in hardware components, but it may also contain faults itself. In both cases, if the operating system does not adequately deal with these issues, they can affect all running applications as well.

Within the operating system, we focus on the software components that are responsible for everything related to storage, from files to storage media. These components are collectively called the **storage stack**. Especially in the storage stack, any problems can easily lead to both application failures and data loss. However, the traditional storage stack found in most operating systems today does not deal well with many problems. This is in part due to the fact that reliability improvements always come with additional overhead, for example in terms of performance loss and higher resource usage. Given the low probability with which these problems occur, it is often hard to justify such overhead.

In this dissertation, we explore ways to improve the overall reliability of the storage stack, while keeping the overhead of these improvements acceptably low. We consider four reliability threats: *storage device failures*, where a storage device may not only stop working altogether but also behave in various erroneous ways; *whole-system failures*, where the entire computer system shuts down unexpectedly; *memory corruption*, where contents of volatile memory are changed as a result of cosmic radiation; and *software bugs*, where part of the storage stack itself misbehaves due to a programming error.

As the basis of our reliability improvements, we subject the storage stack to a fundamental rearrangement. The heart of the traditional storage stack consists of a file system layer and a software-implemented RAID (Redundant Array of Inexpensive Disks) layer. We first split up the traditional file system into three layers. We then move the traditional software-RAID layer between two of those layers, thereby altering the abstraction level on which it operates. The result is a new storage stack arrangement, which we call **Loris**. By design, the Loris stack copes better with storage device failures than the traditional arrangement. Loris also has a number of advantages in other areas.

The Loris stack thus consists of four layers. At the bottom, directly above the operating system's storage hardware driver, is the *physical layer*, which is responsible for the on-device storage layout. On top of it is the *logical layer*, which provides RAID-like redundancy. Next is the *cache layer*, which takes care of caching file data. The topmost Loris layer is the *naming layer*, which manages the file hierarchy. The naming layer runs below the standard Virtual File System (VFS) layer, which provides the operating system's interface for file-related requests from applications.

In subsequent projects, we further strengthen the Loris stack against the reliability threats. We start from a prototype implementation of the Loris storage stack in the MINIX3 microkernel operating system, where the four Loris layers–just like the storage hardware driver, the VFS layer, and all applications–run as separate *userspace processes*. The MINIX3 environment already provides part of the solution for dealing with software bugs. Software bugs typically result in a *crash* of the process in which they manifest themselves. MINIX3 offers a basic facility to restart operating system processes after a crash, after which it only takes a recovery procedure to recover the process to its pre-crash state. The design of the actual recovery procedures for the Loris processes is part of our research.

We develop a number of individual reliability improvements. We use a layer-oriented approach, looking at one or more threats per layer at once. For the physical and logical layers, we show how a system of *recovery points* can form the basis for recovery from both whole-system failures and crashes in these two layers. In the cache layer, we show how *checksums* can both aid in detection of memory corruption and self-recovery of the cache layer after a crash. For the naming layer, we design a system of *transactions* to allow easy recovery from crashes in this layer. In all cases, whenever possible, we ensure that the running applications are not exposed to the problems that occur.

In addition, we consider software bugs in the VFS layer. It would be difficult to implement low-overhead recovery support for this layer, and thus, we opt for another approach. We rework the operating system such that each application gets its own private copy of the VFS layer as well as the Loris naming layer. Then, whenever either of these two layers fails irrecoverably, only one application is affected. We demonstrate that further extending this idea yields a new form of computer system *virtualization*. This new virtualization approach, where a virtualizing version of Loris plays a key role, is a middle-ground alternative to the currently dominant two types of virtualization, providing advantages beyond just reliability.

Finally, we combine all our individual solutions into a reliable, virtualizing operating system storage stack. We evaluate the resulting reliability and performance, and conclude that our work indeed adds significant reliability with acceptably low overhead, thus proving the feasibility of our ideas. We reach a number of additional research conclusions. We find that it is useful to consider more than one reliability threat at once, as the same building blocks can then be reused to address multiple threats. We also find that by focusing only on the storage stack, we are able to leverage semantic knowledge about the storage stack in several beneficial ways. Specifically for the problem of software bugs, we determine the ideal level of decomposition of the storage stack into layers and we draw a substantiated comparison between our individual recovery systems.

Samenvatting

Titel: Constructie van een betrouwbaar opslagsysteem

Een computersysteem kan met diverse problemen in zijn onderdelen te maken krijgen, zowel in hardware als in software. In het algemeen treden zulke problemen niet vaak op, maar als ze niet correct worden aangepakt, kunnen ze leiden tot het falen van het hele computersysteem. Zo'n grotere storing kan niet alleen een onderbreking in de normale werking van het systeem veroorzaken, maar ook leiden tot gegevensverlies, en uiteindelijk zelfs tot financiële schade voor de eigenaar van het computersysteem. Zulke allesomvattende storingen zijn te voorkomen door ervoor te zorgen dat het computersysteem beter kan omgaan met de problemen in de afzonderlijke componenten, waarmee de **betrouwbaarheid** van het computersysteem wordt vergroot.

Het kader van dit onderzoek is het verbeteren van de betrouwbaarheid van computersystemen door middel van aanpassingen in de software. Ons algemene werkgebied is het *besturingssysteem*, doorgaans de onderste laag van software. Eén van de belangrijkste taken van het besturingssysteem is het verzorgen van de interactie met de hardware, ten behoeve van de applicaties die bovenop het besturingssysteem draaien. Als zodanig is het besturingssysteem van bijzonder belang voor betrouwbaarheid. Niet alleen wordt het besturingssysteem als eerste blootgesteld aan problemen in de hardwarecomponenten, maar kan het zelf ook fouten bevatten. In beide gevallen kan inadequate foutafhandeling in het besturingssysteem tevens negatieve gevolgen hebben voor alle draaiende applicaties.

Binnen het besturingssysteem concentreren we ons op de softwarecomponenten die verantwoordelijk zijn voor alles wat gerelateerd is aan gegevensopslag, van bestanden tot opslagmedia. Deze componenten worden samen het **opslagsysteem** genoemd. Met name in het opslagsysteem kunnen problemen en vergissingen al snel

leiden tot het falen van applicaties en tot gegevensverlies. Het gebruikelijke opslagsysteem dat aanwezig is in de meeste besturingssystemen kan echter veel problemen niet goed aan. Dit valt mede te verklaren doordat verbeteringen in betrouwbaarheid altijd *overhead* met zich meebrengen, bijvoorbeeld in de vorm van verlies aan snelheid en hoger gebruik van hardwarebronnen. De lage kans dat dergelijke problemen ook echt optreden maakt het lastig om deze overhead te rechtvaardigen.

In deze dissertatie doen we onderzoek naar manieren om de algemene betrouwbaarheid van het opslagsysteem te verbeteren, zonder dat de extra overhead daarvoor te hoog wordt. We bekijken vier bedreigingen voor de betrouwbaarheid van het opslagsysteem: *het falen van opslagapparaten*, waarbij een opslagmedium niet alleen geheel kan uitvallen maar ook op diverse manieren ongewenst verdrag kan vertonen; *systeemuitval*, waarbij het gehele computersysteem onverwachts wordt uitgeschakeld; *geheugencorruptie*, waarbij kosmische straling zorgt voor veranderingen in het werkgeheugen; en *software-bugs*, waarbij een gedeelte van het opslagsysteem zelf door een programmeerfout onbedoeld gedrag vertoont.

Als basis van onze betrouwbaarheidsverbeteringen onderwerpen we het opslagsysteem aan een fundamentele herstructurering. De kern van het gebruikelijke opslagsysteem bestaat uit een bestandssysteem-laag en een software-gebaseerde RAID-laag (Redundant Array of Inexpensive Disks). We splitsen eerst het gebruikelijke bestandssysteem op in drie nieuwe lagen. Vervolgens schuiven we de gebruikelijke software-RAID laag tussen twee van deze lagen, waarmee we het abstractieniveau veranderen waarop deze laag opereert. Het resultaat is een nieuwe rangschikking van het opslagsysteem, die we **Loris** noemen. Het ontwerp van Loris zorgt ervoor dat het beter omgaat met het falen van opslagapparaten dan de gebruikelijke schikking. Loris biedt tevens een aantal voordelen op andere vlakken.

Het Loris-systeem bestaat aldus uit vier lagen. De onderste laag, direct boven het opslagmedia-sturingsprogramma van het besturingssysteem, is de *fysieke laag*, die verantwoordelijk is voor de indeling van gegevens in de opslagmedia. Daarbovenop bevindt zich de *logische laag*, die RAID-achtige redundantie biedt. De volgende laag is de *cache-laag*, die het *cachen* van bestandsgegevens verzorgt. De bovenste laag van Loris is de *naamgevingslaag*, die de bestandshiërarchie beheert. De naamgevingslaag draait onder de standaard *Virtual File System* (VFS) laag. De VFS-laag fungeert als het doorgeefluik van het besturingssysteem voor bestandsgerelateerde verzoeken die afkomstig zijn van applicaties.

In het vervolg van het onderzoek versterken we het Loris systeem verder tegen de genoemde betrouwbaarheidsbedreigingen. We doen dit op basis van een prototype van het Loris-opslagsysteem dat we implementeren in het MINIX3 microkernel-besturingssysteem. Op dit besturingssysteem draaien de vier lagen van Loris – evenals het opslagmedia-sturingsprogramma, de VFS-laag en alle applicaties – als aparte *user-space processen*. De MINIX3-omgeving biedt meteen al een deel van de oplossing voor het omgaan met software-bugs. In het algemeen leiden software-bugs namelijk tot een *crash* van het proces waarin ze optreden. MINIX3 biedt de basisfaciliteit om gecrashte besturingssysteem-processen te herstarten, waarna er slechts een

herstelprocedure nodig is om het proces terug te brengen in de toestand van voor de crash. Het ontwerp van de herstelprocedures voor de Loris-processen is onderdeel van ons onderzoek.

We ontwikkelen een aantal individuele betrouwbaarheidsverbeteringen. Hierbij kijken we laag voor laag naar één bedreiging of verschillende tegelijk. Voor de fysieke en logische laag tonen we aan hoe een systeem van *herstelpunten* de basis kan vormen voor herstel na systeemuitval of crashes in deze twee lagen. In de cache-laag laten we zien hoe *controlegetallen* kunnen helpen zowel bij het opsporen van geheugencorruptie als met zelf-herstel van de cache-laag na een crash. Voor de naamgevingslaag ontwerpen we een systeem van *transacties* om eenvoudig herstel na crashes in deze laag mogelijk te maken. In alle gevallen zorgen we dat, waar mogelijk, de draaiende applicaties niet blootgesteld worden aan opgetreden problemen.

Verder bekijken we software-bugs in de VFS-laag. Het is moeilijk om voor deze laag met geringe extra overhead herstelondersteuning aan te brengen, en daarom kiezen we hier voor een andere aanpak. We passen het besturingssysteem zo aan dat elke applicatie zijn eigen privékopie krijgt van zowel de VFS-laag als de naamgevingslaag. Als één van deze twee lagen dan onherstelbaar crasht, wordt hierdoor slechts een enkele applicatie getroffen. We tonen aan dat als we dit basisidee verder uitbreiden, dit ons een nieuwe vorm van computersysteem-**virtualisatie** oplevert. Deze nieuwe virtualisatie-aanpak, waarin een virtualiserende versie van Loris een hoofdrol speelt, is een middenweg-alternatief voor de momenteel dominante twee vormen van virtualisatie, met meer voordelen dan alleen voor de betrouwbaarheid.

Tenslotte combineren we al onze afzonderlijke oplossingen tot een betrouwbaar, virtualiserend opslagsysteem voor besturingssystemen. We evalueren de resulterende betrouwbaarheid en prestaties, en concluderen dat ons systeem inderdaad met acceptabel lage overhead een aanzienlijke hoeveelheid betrouwbaarheid toevoegt. Daarmee bewijzen we de haalbaarheid van onze ideeën. We komen tot een aantal aanvullende onderzoeksconclusies. We concluderen dat het nuttig is om meer dan één betrouwbaarheidsbedreiging tegelijk in ogenschouw te nemen, omdat dezelfde bouwstenen dan hergebruikt kunnen worden om meerdere bedreigingen aan te pakken. We concluderen ook dat door onze focus uitsluitend te richten op het opslagsysteem, we tevens in staat zijn om semantische kennis over het opslagsysteem op meerdere manieren nuttig te gebruiken. Specifiek voor het probleem van softwarebugs bepalen we het ideale niveau van opsplitsing van het opslagsysteem in lagen, en maken we een onderbouwde vergelijking tussen onze afzonderlijke herstelsystemen.